Table of Contents

前言

首先声明，我和张晓翀都不是算法牛人，确切的说应该是算法的门外汉，小白一个。所以我们为了撬开算法的大门，各自刷完了一遍LeetCode的题目，这其中碰到了很多困难，当然也少不了用了Google以及参考了别人的代码。

做完一遍下来，陡然发现，很多题目还是忘记了，再次碰到又不知道如何下手，其实这就是典型的没有理解，掌握透。所以我们决定，应该好好的将自己做题的思路记录下来，一个知识点，自己能弄懂，写出来让大家都明白，同时能做到举一反三，触类旁通，那么在一定程度上面才算是真的掌握了。

于是就有了现在准备开始的这本书：《LeetCode题解》，用来记录我们刷LeetCode题目时候的心酸历史。我们保证，书中的代码一定通过了当时LeetCode的测试，虽然后续可能因为LeetCode测试条件的改变导致某些解题无法通过，但我们会实时的跟进。

编程语言使用C++，代码风格上面并没有强制的采用什么编码规范，毕竟是算法解题，只需要代码清晰易懂就可以了。

我们准备按照LeetCode的题型分类来组织章节，譬如Array，Hash Table等，而对每个章节里面的题目，通常采用由易到难的方式进行说明。采用这种方式，能让我们在短时间内快速学习掌握某一类知识，同时也便于讲解说明。

当然，除了LeetCode现有的题目，我们也希望在每个章节加入相关的扩展知识，这需要我们参考大量现有的算法书籍，鉴于个人精力时间有限，可能并不会完全实施。

最后，我们非常欢迎大家的反馈（前提是有人看我们的东西）。如果你有任何的意见建议，欢迎在Github的issue里面提出，或者直接与我们联系。

Thanks Contributor

- 陈心宇 collectchen@gmail.com
- 张晓翀 xczhang07@gmail.com

Maintainer

- SiddonTang siddontang@gmail.com

Array

Remove Element

> Given an array and a value, remove all instances of that > value in place and return the new length.
>
> The order of elements can be changed. It doesn't matter what you leave beyond the new length.

作为开胃菜，我当然选取了最容易的一道题目，在一个数组里面移除指定value，并且返回新的数组长度。这题唯一需要注意的地方在于 `in place` ，不能新建另一个数组。

方法很简单，使用两个游标i，j，遍历数组，如果碰到了value，使用j记录位置，同时递增i，直到下一个非value出现，将此时i对应的值复制到j的位置上，增加j，重复上述过程直到遍历结束。这时候j就是新的数组长度。

代码如下：

```
class Solution {
public:
    int removeElement(int A[], int n, int elem) {
        int i = 0;
        int j = 0;
        for(i = 0; i < n; i++) {
            if(A[i] == elem) {
                continue;
            }

            A[j] = A[i];
            j++;
        }

        return j;
    }
};
```

举一个最简单的例子，譬如数组为1，2，2，3，2，4，我们需要删除2，首先初始化i和j为0，指向第一个位置，因为第一个元素为1，所以A[0] = A[0]，i和j都加1，而第二个元素为2，我们递增i，直到碰到3，此时A[1] = A[3]，也就是3，递增i和j，这时候下一个元素又是2，递增i，直到碰到4，此时A[2] = A[5]，也就是4，再次递增i和j，这时候数组已经遍历完毕，结束。这时候j的值为3，刚好就是新的数组的长度。

Remove Duplicates from Sorted Array

> Given a sorted array, remove the duplicates in place such that > each element appear only once and return the new length.
>
> Do not allocate extra space for another array, you must do this in place with constant memory.
>
> For example, Given input array A = [1,1,2],
>
> Your function should return length = 2, and A is now [1,2].

这道题目与前一题Remove Element比较类似。但是在一个排序好的数组里面删除重复的元素。

首先我们需要知道，对于一个排好序的数组来说，`A[N + 1] >= A[N]`，我们仍然使用两个游标i和j来处理，假设现在i = j + 1，如果A[i] == A[j]，那么我们递增i，直到A[i] != A[j]，这时候我们再设置A[j + 1] = A[i]，同时递增i和j，重复上述过程直到遍历结束。

代码如下：

```
class Solution {
public:
    int removeDuplicates(int A[], int n) {
        if(n == 0) {
            return 0;
        }

        int j = 0;
        for(int i = 1; i < n; i++) {
            if(A[j] != A[i]) {
                A[++j] = A[i];
            }
        }
        return j + 1;
    }
};
```

譬如一个数组为1，1，2，3，首先i = 1，j = 0，这时候A[i] = A[j]，于是递增i，碰到2，不等于1，此时设置A[j + 1] = A[i]，也就是A[1] = A[2]，递增i和j为3和1，这时候A[3] != A[1]，设置A[j + 1] = A[i]，也就是A[2] = A[3]，再次递增，遍历结束。这时候新的数组长度就为2 + 1，也就是3。

Remove Duplicates from Sorted Array II

> Follow up for "Remove Duplicates": What if duplicates are allowed at most twice?
>
> For example, Given sorted array A = [1,1,1,2,2,3],
>
> Your function should return length = 5, and A is now [1,1,2,2,3].

紧接着上一题，同样是移除重复的元素，但是可以允许最多两次重复元素存在。

仍然是第一题的思路，但是我们需要用一个计数器来记录重复的次数，如果重复次数大于等于2，我们会按照第一题的方式处理，如果不是重复元素了，我们将计数器清零。

代码如下：

```
class Solution {
public:
    int removeDuplicates(int A[], int n) {
        if(n == 0) {
            return 0;
        }

        int j = 0;
        int num = 0;
        for(int i = 1; i < n; i++) {
            if(A[j] == A[i]) {
                num++;
                if(num < 2) {
                    A[++j] = A[i];
                }
            } else {
                A[++j] = A[i];
                num = 0;
            }
        }
        return j + 1;
    }
};
```

Plus One

> Given a non-negative number represented as an array of digits, plus one to the number.
>
> The digits are stored such that the most significant digit is at the head of the list.

这道题目很简单，就是考的加法进位问题。直接上代码：

```
class Solution {
public:
    vector<int> plusOne(vector<int> &digits) {
        vector<int> res(digits.size(), 0);
        int sum = 0;
        int one = 1;
        for(int i =  digits.size() - 1; i >= 0; i--) {
            sum = one + digits[i];
            one = sum / 10;
            res[i] = sum % 10;
        }

        if(one > 0) {
            res.insert(res.begin(), one);
        }
        return res;
    }
};
```

Pascal's Triangle

> Given numRows, generate the first numRows of Pascal's triangle.
>
> For example, given numRows = 5, Return
>
> ```
> [
>      [1],
>     [1,1],
>    [1,2,1],
>   [1,3,3,1],
>  [1,4,6,4,1]
> ]
> ```

要得到一个帕斯卡三角，我们只需要找到规律即可。

- 第k层有k个元素
- 每层第一个以及最后一个元素值为1
- 对于第k（k > 2）层第n（n > 1 && n < k）个元素A[k][n]，A[k][n] = A[k-1][n-1] + A[k-1][n]

知道了上面的规律，就很好做了，我们使用一个二维数组来存储整个三角，代码如下：

```
class Solution {
public:
    vector<vector<int> > generate(int numRows) {
        vector<vector<int> > vals;
        vals.resize(numRows);

        for(int i = 0; i < numRows; i++) {
            vals[i].resize(i + 1);
            vals[i][0] = 1;
            vals[i][vals[i].size() - 1] = 1;
            for(int j = 1; j < vals[i].size() - 1; j++) {
                vals[i][j] = vals[i - 1][j - 1] + vals[i
- 1][j];
            }
        }

        return vals;
    }
};
```

Pascal's Triangle II

> Given an index k, return the kth row of the Pascal's triangle.
>
> For example, given k = 3, Return [1,3,3,1].

不同于上一题，这里我们仅仅需要得到的第k层的集合，但只能使用O(k)的空间。所以不能用前面二维数组的方式，只能使用一位数组滚动计算。

在第一题里面，我们知道，帕斯卡三角的计算公式是这样的，A[k][n] = A[k-1][n-1] + A[k-1][n]。

假设现在数组存放的第3层的数据，[1, 3, 3, 1]，如果我们需要计算第4层的数据，如果我们从前往后计算，譬如A[4][2]= A[3][1] + A[3][2]，也就是4，但是因为只有一个数组，所以需要将4这个值覆盖到2这个位置，那么我们计算A[4][3]的时候就会出现问题了，因为这时候A[3][2]不是3，而是4了。

为了解决这个问题，我们只能从后往前计算，仍然是上面那个例子，我们实现计算A[4][3] = A[3][2] + A[3][3]，也就是6，我们将6直接覆盖到3这个位置，但不会影响我们计算A[4][2]，因为A[4][2] = A[3][1] + A[3][2]，已经不会涉及到3这个位置了。

理解了如何计算，代码就很简单了：

```
class Solution {
public:
    vector<int> getRow(int rowIndex) {
        vector<int> vals;

        vals.resize(rowIndex + 1, 1);

        for(int i = 0; i < rowIndex + 1; ++i) {
            for(int j = i - 1; j >= 1; --j) {
                vals[j] = vals[j] + vals[j - 1];
            }

        }

        return vals;
    }
};
```

Merge Sorted Array

> Given two sorted integer arrays A and B, merge B into A as one sorted array.
>
> Note: You may assume that A has enough space (size that is greater or equal to m + n) to hold additional elements from B. The number of elements initialized in A and B are m and n respectively.

A和B都已经是排好序的数组，我们只需要从后往前比较就可以了。

因为A有足够的空间容纳A + B，我们使用游标i指向m + n - 1，也就是最大数值存放的地方，从后往前遍历A，B，谁大就放到i这里，同时递减i。

代码如下：

```
class Solution {
public:
    void merge(int A[], int m, int B[], int n) {
        int i = m + n - 1;
        int j = m - 1;
        int k = n - 1;

        while(i >= 0) {
            if(j >= 0 && k >= 0) {
                if(A[j] > B[k]) {
                    A[i] = A[j];
                    j--;
                } else {
                    A[i] = B[k];
                    k--;
                }
            } else if(j >= 0) {
                A[i] = A[j];
                j--;
            } else if(k >= 0) {
                A[i] = B[k];
                k--;
            }
            i--;
        }
    }
};
```

2Sum

> Given an array of intergers, find two numbers such that they add up to a specific target number. The function twoSum should return indices of the two numbers such that they add up to the target, where index1 must be less than index2 Please note that your returned answers (both index1 and index2) are not zero-based.
>
> You may assume that each input would have exactly one solution.
>
> Input: numbers={2, 7, 11, 15}, target=9 Output: index1=1, index2=2

题目翻译： 这道题目的意思是给定一个数组和一个值，让求出这个数组中两个值的和等于这个给定值的坐标。输出是有要求的，1， 坐标较小的放在前面，较大的放在后面。2， 这俩坐标不能为零。

题目分析： 第一步： 我们要分析题意，其中有三个关键点：

1. 求出来的坐标值要按序排列。
2. 这两个坐标不能从零开始。
3. 这道题目假设是只有一组答案符合要求，这样降低了我们解题的难度。

根据题目我们可以得到以下信息：

1. 我们得到坐标的时候，要根据大小的顺序放入数组。
2. 因为坐标值不能为零，所以我们得到的坐标都要+1。
3. 因为有且只有一组答案符合要求，所以这大大的降低了这道题目的难度，也就是说，我们只要找到符合条件的两个数，存入结果，直接终止程序，返回答案即可。

解题思路： 这道题不是很难，是leetcode最开始的题目，要求很明确，很直接，如果我们用两个for循环，O(n2)的时间复杂度去求解的话，很容易计算出来，但这明显不是面试官需要的答案。brute force只有在你不知道如何优化题目的时候，将就的给出的一个解法。。那么我们能不能用O(n)的时间复杂度去解这道题呢？很显然是可以的，不过，天下没有掉馅饼的事情啦，既然优化了时间复杂度，我们就要牺牲空间复杂度啦。在这里用什么呢？stack？queue？vector？还是hash_map?

对于stack和queue，除了pop外，查找的时间复杂度也是O(n)。明显不是我们所需要的，那么什么数据结构的查找时间复杂度小呢？很显然是 hash_map, 查找的时间复杂度理想情况下是O(1)。所以我们先来考虑hash_map，看看hash_map怎么求解这个问题。

我们可以先把这个数组的所有元素存到hashmap中，一次循环就行， 时间复杂度为O(n),之后对所给数组在进行遍历，针对其中的元素我们只要用another_number = target-numbers[i],之后用hashmap的find function来查找这个值，如果存在的话，在进行后续比较（详见代码），如果不存在的话，继续查找，好啦，思路已经摆在这里了，详见代码吧。

```
class Solution {
public:
    vector<int> twoSum(vector<int> &numbers, int target)
{
        //边角问题，我们要考虑边角问题的处理
        vector<int> ret;
        if(numbers.size() <= 1)
            return ret;
        //新建一个map<key,value> 模式的来存储numbers里面的元素
和index，
        //这里的unordered_map相当于hash_map
        unordered_map<int,int> myMap;
        for(int i = 0; i < numbers.size(); ++i)
            myMap[numbers[i]] = i;
        for(int i = 0; i < numbers.size(); ++i)
        {
            int rest_val = target - numbers[i];
            if(myMap.find(rest_val)!=myMap.end())
            {
                int index = myMap[rest_val];
                if(index == i)
                    continue;      //如果是同一个数字，我们就
pass，是不会取这个值的
                if(index < i)
```

```
                {
                    ret.push_back(index+1);  //这里+1是因
为题目说明白了要non-zero based index
                    ret.push_back(i+1);
                    return ret;
                }
                else
                {
                    ret.push_back(i+1);
                    ret.push_back(index+1);
                    return ret;
                }
            }
        }
    }
};
```

以上解法的要注意的是记得要检查这两个坐标是不是相同的，因为我们并不需要同样的数字。

3Sum

> Given an array S of n integers, are there elements a, b, c in S such that a + b + c = 0? Find all unique triplets in the array which gives the sum of zero.
>
> Note: Elements in a triplet (a,b,c) must be in non-descending order. (ie, a ≤ b ≤ c) The solution set must not contain duplicate triplets.

题目翻译：

给定一个整型数组num，找出这个数组中满足这个条件的所有数字：num[i]+num[j]+num[k] = 0. 并且所有的答案是要和其他不同的，也就是说两个相同的答案是不被接受的。

题目的两点要求：

1. 每个答案组里面的三个数字是要从大到小排列起来的。

2. 每个答案不可以和其他的答案相同。

题目分析：

1. 每一个答案数组triplet中的元素是要求升序排列的。
2. 不能包含重复的答案数组。

解题思路：

1. 根据第一点要求：因为要求每个答案数组中的元素是升序排列的，所以在开头我们要对数组进行排序。

2. 根据第二点要求：因为不能包含重复的答案数组，所以我们要在代码里面做一切去掉重复的操作，对于数组，这样的操作是相同的。最开始我做leetcode的时候是把所有满足条件的答案数组存起来，之后再用map进行处理，感觉那样太麻烦了，所以这次给出的答案是不需要额外空间的。

时间复杂度分析：

对于这道题，因为是要找三个元素，所以怎样都要O(n2)的时间复杂度，目前我没有想出来O(n)时间复杂度的解法。

归根结底，其实这是two pointers的想法，定位其中两个指针，根据和的大小来移动另外一个。解题中所要注意的就是一些细节问题。好了，上代码吧。

```
class Solution {
public:
//constant space version
    vector<vector<int> > threeSum(vector<int> &num) {
        vector<vector<int>> ret;
        //corner case invalid check
        if(num.size() <= 2)
            return ret;

        //first we need to sort the array because we need
the non-descending order
        sort(num.begin(), num.end());

        for(int i = 0; i < num.size()-2; ++i)
```

```
        {
            int j = i+1;
            int k = num.size()-1;
            while(j < k)
            {
                vector<int> curr;   //create a tmp vector
 to store each triplet which satisfy the solution.
                if(num[i]+num[j]+num[k] == 0)
                {
                    curr.push_back(num[i]);
                    curr.push_back(num[j]);
                    curr.push_back(num[k]);
                    ret.push_back(curr);
                    ++j;
                    --k;
                    //this two while loop is used to skip
 the duplication solution
                    while(j < k&&num[j-1] == num[j])
                        ++j;
                    while(j < k&&num[k] == num[k+1])
                        --k;
                }
                else if(num[i]+num[j]+num[k] < 0)  //if
 the sum is less than the target value, we need to move j
 to forward
                    ++j;
                else
                    --k;
            }
            //this while loop also is used to skip the
 duplication solution
            while(i < num.size()-1&&num[i] == num[i+1])
                ++i;
        }
        return ret;
    }
```

```
};
```

根据以上代码，我们要注意的就是用于去除重复的那三个while loop。一些细节问题比如for loop中的i < num.size()-2;因为j和k都在i后面，所以减掉两位。当然如果写成i< num.size(); 也是可以通过测试的，但感觉思路不是很清晰。另外一点，就是不要忘记了corner case check呀。

3Sum Closest

> Given an array S of n integers, find three integers in S such that the sum is closest to a given number, target. Return the sum of the three integers. You man assume that each input would have exactly one solution.

题目翻译：

给定一个整形数组S和一个具体的值，要求找出在这数组中三个元素的和和这个给定的值最小。input只有一个有效答案。

题目要求：

这道题比较直接，也没有什么具体的要求。

题目分析：

1. 最短距离：两个整数的最短距离是0.这点对于这道题比较重要，别忽略。
2. 这道题和3Sum几乎同出一辙，所以方便于解题，我们还是在开头要对数组进行排序，要么没法定位指针移动。

3. 另外，这道题中用到了INT_MAX这个值，这个值和 INT_MIN是相对应的，在很多比较求最大值最小值的情况，经常用这两个变量。

解题思路：

这道题的解题方法和3Sum几乎相同，设定三个指针，固定两个，根据和的大小移动另外一个。属于这道题目自己的东西就是distance比较这块儿，建立一个tmp distance和min distance比较。

时间复杂度分析：

这道题目和3Sum几乎是一个思路，所以时间复杂度为O(n2)。

代码如下：

```
class Solution {
public:

    int threeSumClosest(vector<int> &num, int target) {
        //invalid corner case check
        if(num.size() <= 2)
            return -1;

        int ret = 0;
        //first we suspect the distance between the sum
and the target is the largest number in int
        int distance = INT_MAX;
        sort(num.begin(),num.end());  //sort is needed
        for(int i = 0; i < num.size()-2; ++i)
        {
            int j = i+1;
            int k = num.size()-1;
            while(j < k)
            {
                int tmp_val = num[i]+num[j]+num[k];
                int tmp_distance;
                if(tmp_val < target)
                {
                    tmp_distance = target - tmp_val;
                    if(tmp_distance < distance)
                    {
                        distance = tmp_distance;
                        ret = num[i]+num[j]+num[k];
                    }
                    ++j;
                }
                else if(tmp_val > target)
                {
```

```
                    tmp_distance = tmp_val-target;
                    if(tmp_distance < distance)
                    {
                        distance = tmp_distance;
                        ret= num[i]+num[j]+num[k];
                    }
                    --k;
                }
                else //note: in this case, the sum is 0, 0 means the shortest distance from the sum to the target
                {
                    ret = num[i]+num[j]+num[k];
                    return ret;
                }
            }
        }
        return ret;
    }
};
```

总结：这道题的解决方法主要要注意以下几点：

1. 首先要对数组进行排序。
2. 0是两个数组间最小的距离。

4Sum

> Given an array S of n integers, are there elements a, b, c and d in S such that a+b+c+d = target? Find all unique quadruplets in the array which gives the sume of target.
>
> Note:
>
> 1. Elements in quadruplets (a, b, c, d) must be in non-descending order. (ie, a<=b<=c<=d)
> 2. The solution must not contain duplicates quadruplets.

题目翻译：

给定一个整型数字数组num和一个目标值target，求出数组中所有的组合满足条件： num[a]+num[b]+num[c]+num[d] = target.

并且要满足的条件是：

1. num[a] <= num[b] <= num[c] <= num[d]
2. 答案中的组合没有重复的.

题目分析：

这道题和3Sum几乎同出一辙，只不过是要求四个数字的和，在时间复杂度上要比3Sum高一个数量级。对于两点要求的处理：

1. 首先要对整个数组进行排序，这样得到的答案自然是排序好的.
2. 对于重复答案的处理和3Sum是一摸一样的。

解题思路： 同3Sum.

时间复杂度分析：

这道题的解法，我选择的是空间复杂度为1， 时间复杂度为O(n3).对于这样的问题，如果到了KSum(K>=5), 我觉得可以用hash_map来牺牲空间复杂度换取好一些的时间复杂度.

代码如下：

```
class Solution {
public:
    vector<vector<int> > fourSum(vector<int> &num, int
target) {
        vector<vector<int>> ret;
        if(num.size() <= 3) //invalid corner case check
            return ret;
        sort(num.begin(), num.end()); //cause we need the
result in quadruplets should be non-descending
        for(int i = 0; i < num.size()-3; ++i)
        {
            if(i > 0 && num[i] == num[i-1])
                continue;
```

```
            for(int j = i+1; j < num.size()-2; ++j)
            {
                if(j > i+1 && num[j] == num[j-1])
                    continue;
                int k = j+1;
                int l = num.size()-1;
                while(k < l)
                {
                    int sum =
num[i]+num[j]+num[k]+num[l];
                    if(sum == target)
                    {
                        vector<int> curr; //create a
temporary vector to store the each quadruplets
                        curr.push_back(num[i]);
                        curr.push_back(num[j]);
                        curr.push_back(num[k]);
                        curr.push_back(num[l]);
                        ret.push_back(curr);
                        //the two while loops are used to
skip the duplication solutions
                        do{++k;}
                        while(k<l && num[k] == num[k-1]);
                        do{--l;}
                        while(k<l && num[l] == num[l+1]);
                    }
                    else if(sum < target)
                        ++k;  //we can do this operation
because of we sort the array at the beginning
                    else
                        --l;
                }
            }
        }
        return ret;
    }
```

```
};
```

根据上述代码，我觉得要说明白的一点是:

1. 我用do{}while来代替了while进行重复答案的处理，为什么要这样呢？是因为如果换成了while，leetcode的test sample过不去，报的错误是超出了时间的限制，我认为如果要用while，应该是多进行了++k 还有--l的操作吧。换成了do{}while就可以通过所有的test case.

问题扩展

KSum

> 根据以上的2Sum, 3Sum, 3Sum Cloest， 还有4Sum，我相信只要认真看完每道题的解法的童鞋，都会发现一定的规律，相信这时候会有人想，如果变成KSum问题，我们应该如何求解？这是个很好的想法，下面，我们来看看问题扩展.

首先，对于2Sum，我们用的解法是以空间复杂度来换取时间复杂度，那么，2Sum我们可不可以in place来解？时间复杂度又是多少？ 答案是当然可以，我们可以先sort一遍，之后再扫一变，sort的时间复杂度是O(nlogn)，扫一遍是O(n),因此，这种解法的时间复杂度是O(nlogn), 当然，如果对于要找index，leetcode上的题不能用这个方法，因为我们sort一遍之后，index会发生一些变化。但是我们可以用以下这个function来作为一个Helper function对于K Sum(考虑到如果K > 2, sort一遍数组的时间开销不算是主要的时间开销了):

```
void twoSum(vector<int> &numbers, int begin, int first,
int second, int target, vector<vector<int>>& ret) {
        if(begin >= numbers.size()-1)
             return;
         int b = begin;
         int e = numbers.size()-1;
         while(b < e)
         {
             int rest = numbers[b]+numbers[e];
             if(rest == target)
             {
                 vector<int> tmp_ret;
                 tmp_ret.push_back(first);
                 tmp_ret.push_back(second);
                 tmp_ret.push_back(numbers[b]);
                 tmp_ret.push_back(numbers[e]);
                 ret.push_back(tmp_ret);
                 do{b++;}
                 while(b<e && numbers[b] == numbers[b-1]);
                 do{e--;}
                 while(b<e && numbers[e] == numbers[e+1]);
             }
             else if(rest < target)
                 ++b;
             else
                 --e;
         }
     }
```

给个例子，对于4Sum，我们可以调用这个function，代码如下:

```
class Solution {
public:

    void twoSum(vector<int> &numbers, int begin, int
```

```
first, int second, int target, vector<vector<int>>& ret)
{
       if(begin >= numbers.size()-1)
            return;
        int b = begin;
        int e = numbers.size()-1;
        while(b < e)
        {
            int rest = numbers[b]+numbers[e];
            if(rest == target)
            {
                vector<int> tmp_ret;
                tmp_ret.push_back(first);
                tmp_ret.push_back(second);
                tmp_ret.push_back(numbers[b]);
                tmp_ret.push_back(numbers[e]);
                ret.push_back(tmp_ret);
                do{b++;}
                while(b<e && numbers[b] == numbers[b-1]);
                do{e--;}
                while(b<e && numbers[e] == numbers[e+1]);
            }
            else if(rest < target)
                ++b;
            else
                --e;
        }
    }
    vector<vector<int> > fourSum(vector<int> &num, int
target) {
        vector<vector<int>> ret;
        if(num.size() <= 3) //invalid corner case check
            return ret;
        sort(num.begin(), num.end()); //cause we need the
result in quadruplets should be non-descending
        for(int i = 0; i < num.size()-3; ++i)
```

```
        {
            if(i > 0 && num[i] == num[i-1])
                continue;
            for(int j = i+1; j < num.size()-2; ++j)
            {
                if(j > i+1 && num[j] == num[j-1])
                    continue;
                twoSum(num, j+1, num[i], num[j], target-
(num[i]+num[j]), ret);
            }
        }
        return ret;
    }
};
```

以上解法可以延伸到KSum.不过是相当于对于n-2个数做嵌套循环。这么写出来使得思路清晰，以后遇到了类似问题可以解决。

Find Minimum in Rotated Sorted Array

> Suppose a sorted array is rotated at some pivot unknown to you beforehand.
>
> (i.e., 0 1 2 4 5 6 7 might become 4 5 6 7 0 1 2).
>
> Find the minimum element.
>
> You may assume no duplicate exists in the array.

这题要求在一个轮转了的排序数组里面找到最小值，我们可以用二分法来做。

首先我们需要知道，对于一个区间A，如果A[start] < A[stop]，那么该区间一定是有序的了。

假设在一个轮转的排序数组A，我们首先获取中间元素的值，A[mid]，mid = (start + stop) / 2。因为数组没有重复元素，那么就有两种情况：

- A[mid] > A[start]，那么最小值一定在右半区间，譬如[4,5,6,7,0,1,2]，中间元素为7，7 > 4，最小元素一定在[7,0,1,2]这边，于是我们继续在这个区间查找。
- A[mid] < A[start]，那么最小值一定在左半区间，譬如[7,0,1,2,4,5,6]，这件元素为2，2 < 7，我们继续在[7,0,1,2]这个区间查找。

代码如下:

```
class Solution {
public:
    int findMin(vector<int> &num) {
        int size = num.size();

        if(size == 0) {
            return 0;
        } else if(size == 1) {
            return num[0];
        } else if(size == 2) {
            return min(num[0], num[1]);
        }

        int start = 0;
        int stop = size - 1;

        while(start < stop - 1) {
            if(num[start] < num[stop]) {
                return num[start];
            }

            int mid = start + (stop - start) / 2;
            if(num[mid] > num[start]) {
                start = mid;
            } else if(num[mid] < num[start]) {
                stop = mid;
            }
        }

        return min(num[start], num[stop]);
    }
};
```

Find Minimum in Rotated Sorted Array

> Suppose a sorted array is rotated at some pivot unknown to you beforehand.
>
> (i.e., 0 1 2 4 5 6 7 might become 4 5 6 7 0 1 2).
>
> Find the minimum element.
>
> The array may contain duplicates.

这题跟上题唯一的区别在于元素可能有重复，我们仍然采用上面的方法，只是需要处理mid与start相等这种额外情况。

- A[mid] > A[start]，右半区间查找。
- A[mid] < A[start]，左半区间查找。
- A[mid] = A[start]，出现这种情况，我们跳过start，重新查找，譬如[2,2,2,1]，A[mid] = A[start]都为2，这时候我们跳过start，使用[2,2,1]继续查找。

代码如下:

```
class Solution {
public:
    int findMin(vector<int> &num) {
        int size = num.size();

        if(size == 0) {
            return 0;
        } else if(size == 1) {
            return num[0];
        } else if(size == 2) {
            return min(num[0], num[1]);
        }

        int start = 0;
        int stop = size - 1;

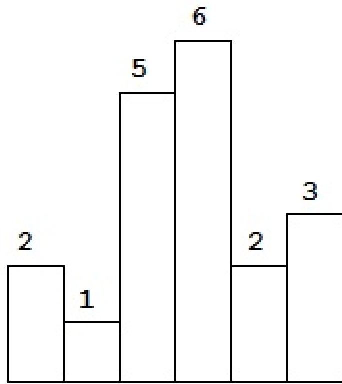
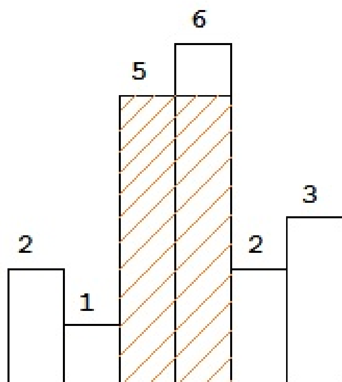
        while(start < stop - 1) {
            if(num[start] < num[stop]) {
                return num[start];
            }

            int mid = start + (stop - start) / 2;
            if(num[mid] > num[start]) {
                start = mid;
            } else if(num[mid] < num[start]) {
                stop = mid;
            } else {
                start++;
            }
        }

        return min(num[start], num[stop]);
    }
};
```

这题需要注意，如果重复元素很多，那么最终会退化到遍历整个数组，而不是二分查找了。

Largest Rectangle in Histogram

> Given n non-negative integers representing the histogram's bar height where the width of each bar is 1, find the area of largest rectangle in the histogram.
>
> 
>
>
> Above is a histogram where width of each bar is 1, given height = [2,1,5,6,2,3].
>
> 
>
>
> The largest rectangle is shown in the shaded area, which has area = 10 unit.
>
> For example, Given height = [2,1,5,6,2,3], return 10.

这道题目算是比较难得一道题目了，首先最简单的做法就是对于任意一个bar，向左向右遍历，直到高度小于该bar，这时候计算该区域的矩形区域面积。对于每一个bar，我们都做如上处理，最后就可以得到最大值了。当然这种做法是O(n2)，铁定过不了大数据集合测试的。

从上面我们直到，对于任意一个bar n，我们得到的包含该bar n的矩形区域里面bar n是最小的。我们使用ln和rn来表示bar n向左以及向右第一个小于bar n的bar的索引位置。

譬如题目中的bar 2的高度为5，它的ln为1，rn为4。包含bar 2的矩形区域面积为(4 - 1 - 1) * 5 = 10。

我们可以从左到右遍历所有bar，并将其push到一个stack中，如果当前bar的高度小于栈顶bar，我们pop出栈顶的bar，同时以该bar计算矩形面积。那么我们如何知道该bar的ln和rn呢？rn铁定就是当前遍历到的bar的索引，而ln则是当前的栈顶bar的索引，因为此时栈顶bar的高度一定小于pop出来的bar的高度。

为了更好的处理最后一个bar的情况，我们在实际中会插入一个高度为0的bar，这样就能pop出最后一个bar并计算了。

代码如下：

```
class Solution {
public:
    int largestRectangleArea(vector<int> &height) {
        vector<int> s;
        //插入高度为0的bar
        height.push_back(0);

        int sum = 0;
        int i = 0;
        while(i < height.size()) {
            if(s.empty() || height[i] > height[s.back()])
{
                s.push_back(i);
                i++;
            } else {
                int t = s.back();
                s.pop_back();
                //这里还需要考虑stack为空的情况
                sum = max(sum, height[t] * (s.empty() ? i
: i - s.back() - 1));
            }
        }

        return sum;
    }
};
```

Maximal Rectangle

> Given a 2D binary matrix filled with 0's and 1's, find the largest rectangle containing all ones and return its area.

这题是一道难度很大的题目，至少我刚开始的时候完全不知道怎么做，也是google了才知道的。

这题要求在一个矩阵里面求出全部包含1的最大矩形面积，譬如这个：

```
0 0 0 0
1 1 1 1
1 1 1 0
0 1 0 0
```

我们可以知道，最大的矩形面积为6。也就是下图中虚线包围的区域。那么我们如何得到这个区域呢？

```
 0   0   0   0
|--------|
|1   1   1 |1
|1   1   1 |0
|--------|
 0   1   0   0
```

对于上面哪一题，我们先去掉最下面的一行，然后就可以发现，它可以转化成一个直方图，数据为[2, 2, 2, 0]，我们认为1就是高度，如果碰到0，譬如上面最右列，则高度为0，而计算这个直方图最大矩形面积就很容易了，我们已经在Largest Rectangle in Histogram处理了。

所以我们可以首先得到每一行的直方图，分别求出改直方图的最大区域，最后就能得到结果了。

代码如下：

```
class Solution {
public:
    int maximalRectangle(vector<vector<char> > &matrix) {
        if(matrix.empty() || matrix[0].empty()) {
            return 0;
        }

        int m = matrix.size();
        int n = matrix[0].size();

        vector<vector<int> > height(m, vector<int>(n,
0));
        for(int i = 0; i < m; i++) {
            for(int j = 0; j < n; j++) {
                if(matrix[i][j] == '0') {
                    height[i][j] = 0;
                } else {
                    height[i][j] = (i == 0) ? 1 :
height[i - 1][j] + 1;
                }
            }
        }

        int maxArea = 0;
        for(int i = 0; i < m; i++) {
            maxArea = max(maxArea,
largestRectangleArea(height[i]));
        }
        return maxArea;
    }

    int largestRectangleArea(vector<int> &height) {
        vector<int> s;
        height.push_back(0);
```

```
        int sum = 0;
        int i = 0;
        while(i < height.size()) {
            if(s.empty() || height[i] > height[s.back()])
{
                s.push_back(i);
                i++;
            } else {
                int t = s.back();
                s.pop_back();
                sum = max(sum, height[t] * (s.empty() ? i
: i - s.back() - 1));
            }
        }

        return sum;
    }
};
```

Palindrome Number

> Determine whether an integer is a palindrome. Do this without extra space.

题目翻译: 给定一个数字，要求判断这个数字是否为回文数字. 比如121就是回文数字，122就不是回文数字.

解题思路: 这道题很明显是一道数学题,计算一个数字是否是回文数字，我们其实就是将这个数字除以10，保留他的余数，下次将余数乘以10，加上这个数字再除以10的余数.

需要注意的点:

1. 负数不是回文数字.
2. 0是回文数字.

时间复杂度: logN

代码如下:

```
class Solution {
public:
    bool isPalindrome(int x) {
        if(x < 0)
            return false;
        else if(x == 0)
            return true;
        else
        {
            int tmp = x;
            int y = 0;
            while(x != 0)
            {
                y = y*10 + x%10;
                x = x/10;
            }
            if(y == tmp)
                return true;
            else
                return false;
        }
    }
};
```

Search a 2D Matrix

> Write an efficient algorithm that searches for a value in an m x n matrix. This matrix has the following properties:
>
> Integers in each row are sorted from left to right. The first integer of each row is greater than the last integer of the previous row. For example,
>
> Consider the following matrix:

```
[
  [1,   3,  5,  7],
  [10, 11, 16, 20],
  [23, 30, 34, 50]
]
```

题目翻译: 给定一个矩阵和一个特定值，要求写出一个高效的算法在这个矩阵中快速的找出是否这个给定的值存在. 但是这个矩阵有以下特征.

1. 对于每一行，数值是从左到右从小到大排列的.
2. 对于每一列，数值是从上到下从小到大排列的.

题目解析: 对于这个给定的矩阵，我们如果用brute force解法，用两个嵌套循环，O(n2)便可以得到答案.但是我们需要注意的是这道题已经给定了这个矩阵的两个特性，这两个特性对于提高我们算法的时间复杂度有很大帮助，首先我们给出一个O(n)的解法，也就是说我们可以固定住右上角的元素，根据递增或者递减的规律，我们可以判断这个给定的数值是否存在于这个矩阵当中.

```
class Solution {
public:
    bool searchMatrix(vector<vector<int> > &matrix, int
target) {
        /* we set the corner case as below:
           1, if the row number of input matrix is 0, we
set it false
           2, if the colomun number of input matrix is 0,
we set it false*/
        if(matrix.size() == 0)
            return false;
        if(matrix[0].size() == 0)
            return false;
        int rowNumber = 0;
        int colNumber = matrix[0].size()-1;
        while(rowNumber < matrix.size() && colNumber >=
0)
        {
            if(target < matrix[rowNumber][colNumber])
                --colNumber;
            else if(target > matrix[rowNumber]
[colNumber])
                ++rowNumber;
            else
                return true;
        }
        return false;
    }
};
```

Search for a Range

> Given a sorted array of integers, find the starting and ending position of a given target value.
>
> Your algorithm's runtime complexity must be in the order of O(log n).
>
> If the target is not found in the array, return [-1, -1].
>
> For example,
>
> Given [5, 7, 7, 8, 8, 10] and target value 8,
>
> return [3, 4].

这题要求在一个排好序可能有重复元素的数组里面找到包含某个值的区间范围。要求使用O(log n)的时间，所以我们采用两次二分查找。首先二分找到第一个该值出现的位置，譬如m，然后在[m, n)区间内第二次二分找到最后一个该值出现的位置。代码如下：

```
class Solution {
public:
    vector<int> searchRange(int A[], int n, int target) {
       if(n == 0) {
            return vector<int>({-1, -1});
        }

        vector<int> v;
        int low = 0;
        int high = n - 1;
        //第一次二分找第一个位置
        while(low <= high) {
            int mid = low + (high - low) / 2;
            if(A[mid] >= target) {
                high = mid - 1;
            } else {
                low = mid + 1;
```

```
                }
            }

            if(low < n && A[low] == target) {
                v.push_back(low);
            } else {
                return vector<int>({-1, -1});
            }

            low = low;
            high = n - 1;
            //从第一个位置开始进行第二次二分，找最后一个位置
            while(low <= high) {
                int mid = low + (high - low) / 2;
                if(A[mid] <= target) {
                    low = mid + 1;
                } else {
                    high = mid - 1;
                }
            }

            v.push_back(high);
            return v;
        }
};
```

Search Insert Position

> Given a sorted array and a target value, return the index if the target is found. If not, return the index where it would be if it were inserted in order.
>
> You may assume no duplicates in the array.
>
> Here are few examples.
>
> [1,3,5,6], 5 → 2
>
> [1,3,5,6], 2 → 1
>
> [1,3,5,6], 7 → 4
>
> [1,3,5,6], 0 → 0

这题要求在一个排好序的数组查找某值value，如果存在则返回对应index，不存在则返回能插入到数组中的index（保证数组有序）。

对于不存在的情况，我们只需要在数组里面找到最小的一个值大于value的index，这个index就是我们可以插入的位置。譬如[1, 3, 5, 6]，查找2，我们知道3是最小的一个大于2的数值，而3的index为1，所以我们需要在1这个位置插入2。如果数组里面没有值大于value，则插入到数组末尾。

我们采用二分法解决：

```
class Solution {
public:
    int searchInsert(int A[], int n, int target) {
        int low = 0;
        int high = n - 1;

        while(low <= high) {
            int mid = low + (high - low) / 2;
            if(A[mid] == target) {
                return mid;
            } else if(A[mid] < target) {
                low = mid + 1;
            } else {
                high = mid - 1;
            }
        }

        return low;
    }
};
```

Find Peak Element

> A peak element is an element that is greater than its neighbors.
>
> Given an input array where num[i] ≠ num[i+1], find a peak element and return its index.
>
> The array may contain multiple peaks, in that case return the index to any one of the peaks is fine.
>
> You may imagine that num[-1] = num[n] = -∞.
>
> For example, in array [1, 2, 3, 1], 3 is a peak element and your function should return the index number 2.

这题要求我们在一个无序的数组里面找到一个peak元素，所谓peak，就是值比两边邻居大就行了。

对于这题，最简单地解法就是遍历数组，只要找到第一个元素，大于两边就可以了，复杂度为O(N)。但这题还可以通过二分来做。

首先我们找到中间节点mid，如果大于两边返回当前index就可以了，如果左边的节点比mid大，那么我们可以继续在左半区间查找，这里面一定存在一个peak，为什么这么说呢？假设此时的区间范围为[0, mid - 1]， 因为num[mid - 1]一定大于num[mid]了，如果num[mid - 2] <= num[mid - 1]，那么num[mid - 1]就是一个peak。如果num[mid - 2] > num[mid - 1]，那么我们就继续在[0, mid - 2]区间查找，因为num[-1]为负无穷，所以最终我们绝对能在左半区间找到一个peak。同理右半区间一样。

代码如下：

```
class Solution {
public:
    int findPeakElement(const vector<int> &num) {
        int n = num.size();
        if(n == 1) {
            return 0;
        }

        int start = 0;
        int end = n - 1;
        int mid = 0;

        while(start <= end) {
            mid = start + (end - start) / 2;
            if((mid == 0 || num[mid] >= num[mid - 1]) &&
               (mid == n - 1 || num[mid] >= num[mid +
1])) {
                    return mid;
            }else if(mid > 0 && num[mid-1] > num[mid]) {
                end = mid - 1;
            } else {
                start = mid + 1;
            }
        }
        return mid;
    }
};
```

Bit Manipulation

Missing Number

> Given an array containing n distinct numbers taken from 0, 1, 2, ..., n, find the one that is missing from the array.
>
> For example, Given nums = `[0, 1, 3]` return `2` .
>
> Note: Your algorithm should run in linear runtime complexity. Could you implement it using only constant extra space complexity?

题目翻译: 从0到n之间取出n个不同的数，找出漏掉的那个。 注意：你的算法应当具有线性的时间复杂度。你能实现只占用常数额外空间复杂度的算法吗？

题目分析: 最直观的思路是对数据进行排序，然后依次扫描，便能找出漏掉的数字，但是基于比较的排序算法的时间复杂度至少是 `nlog(n)` ，不满足题目要求。

一种可行的具有线性时间复杂度的算法是求和。对0到n求和，然后对给出的数组求和，二者之差即为漏掉的数字。但是这种方法不适用于0是漏掉的数字的情况，因为此时两个和是相同的。（或者也能由此得出漏掉的数字是0）

从CPU指令所耗费的时钟周期来看，比加法更高效率的运算是异或(XOR)运算。本题的标签里有位运算，暗示本题可以用位运算的方法解决。

异或运算的一个重要性质是，相同的数异或得0，不同的数异或不为0，且此性质可以推广到多个数异或的情形。本题的解法如下，首先将0到n这些数进行异或运算，然后对输入的数组进行异或运算，最后将两个结果进行异或运算，结果便是漏掉的数字，因为其他数字在两个数组中都是成对出现的，异或运算会得到0。

时间复杂度：O(n) 空间复杂度：O(1)

代码如下:

```
class Solution {
public:
    int missingNumber(vector<int>& nums) {
        int x = 0;
        for (int i = 0; i <= nums.size(); i++) x ^= i;
        for (auto n : nums) x ^= n;
        return x;
    }
};
```

Power of Two

> Given an integer, write a function to determine if it is a power of two.

题目翻译: 给出一个整数，判断它是否是2的幂。

题目分析: 2的整数次幂对应的二进制数只含有0个或者1个1，所以我们要做的就是判断输入的数的二进制表达形式里是否符合这一条件。 有一种corner case需要注意，当输入的数为负数的时候，一定不是2的幂。

时间复杂度：O(n) 空间复杂度：O(1)

代码如下:

```
class Solution {
public:
    bool isPowerOfTwo(int n) {
       if (n < 0) return false;
       bool hasOne = false;
       while (n > 0) {
           if (n & 1) {
               if (hasOne) {
                   return false;
               }
               else {
                   hasOne = true;
               }
           }
           n >>= 1;
       }
       return hasOne;
    }
};
```

Number of 1 Bits

> Write a function that takes an unsigned integer and returns the number of '1' bits it has (also known as the Hamming weight). For example, the 32-bit integer `11` has binary representation `00000000000000000000000000001011`, so the function should return 3.

题目翻译: 给出一个整数，求它包含二进制1的位数。例如，32位整数 `11` 的二进制表达形式是 `00000000000000000000000000001011` ，那么函数应该返回3。

题目分析: 设输入的数为n，把n与1做二进制的与(AND)运算，即可判断它的最低位是否为1。如果是的话，把计数变量加一。然后把n向右移动一位，重复上述操作。当n变为0时，终止算法，输出结果。

时间复杂度：O(n) 空间复杂度：O(1)

代码如下:

```
class Solution {
public:
    int hammingWeight(uint32_t n) {
        int count = 0;
        while (n > 0) {
            count += n & 1;
            n >>= 1;
        }
        return count;
    }
};
```

Tree

树是一种重要的非线性数据结构，广泛地应用于计算机技术的各个领域。采用树可以实现一些高效地查找算法，例如数据库系统中用到的红黑树等。

树本身的定义是递归的，因此很多涉及到树的算法通常都可以用递归的方式来实现。然而递归算法在数据量较大的时候效率很低，所以通常会将递归改写成迭代算法。

涉及到树的题目主要包括树的遍历，平衡二叉树，查找二叉树等。

Maximum Depth of Binary Tree

> Given a binary tree, find its maximum depth.
>
> The maximum depth is the number of nodes along the longest path from the root node down to the farthest leaf node.

这题要求我们求出一个二叉树最大深度，也就是从根节点到最远的叶子节点的距离。

对于这题，我们只需要递归遍历二叉树，达到一个叶子节点的时候，记录深度，我们就能得到最深的深度了。

代码如下：

```
class Solution {
public:
    int num;
    int maxDepth(TreeNode *root) {
        if(!root) {
            return 0;
        }

        //首先初始化num为最小值
        num = numeric_limits<int>::min();
        travel(root, 1);
        return num;
    }

    void travel(TreeNode* node, int level) {
        //如果没有左子树以及右子树了，就到了叶子节点
        if(!node->left && !node->right) {
            num = max(num, level);
            return;
        }

        if(node->left) {
            travel(node->left, level + 1);
        }

        if(node->right) {
            travel(node->right, level + 1);
        }
    }
};
```

Minimum Depth of Binary Tree

> Given a binary tree, find its minimum depth.
>
> The minimum depth is the number of nodes along the shortest path from the root node down to the nearest leaf node.

这题跟上题几乎一样，区别在于需要求出根节点到最近的叶子节点的深度，我们仍然使用遍历的方式。

代码如下：

```
class Solution {
public:
    int n;
    int minDepth(TreeNode *root) {
        if(!root) {
            return 0;
        }

        //初始化成最大值
        n = numeric_limits<int>::max();
        int d = 1;

        depth(root, d);
        return n;
    }

    void depth(TreeNode* node, int& d) {
        //叶子节点，比较
        if(!node->left && !node->right) {
            n = min(n, d);
            return;
        }

        if(node->left) {
            d++;
            depth(node->left, d);
            d--;
```

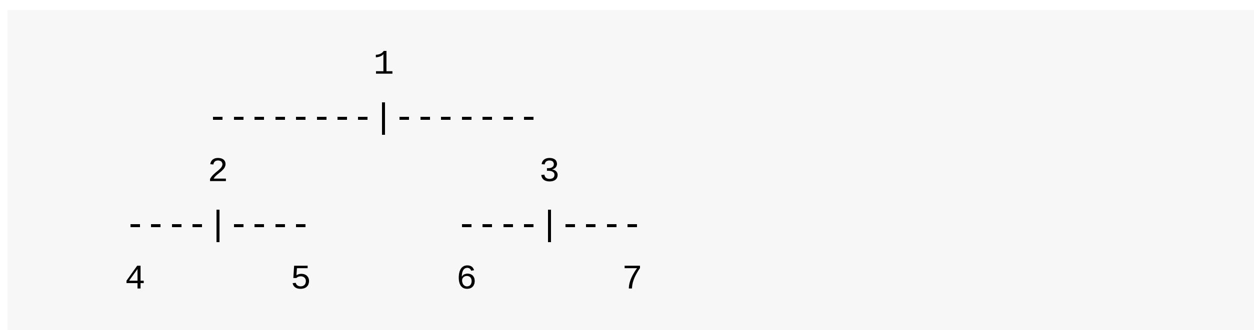
```
            }

            if(node->right) {
                d++;
                depth(node->right, d);
                d--;
            }
        }
};
```

Construct Binary Tree from Inorder and Postorder Traversal

> Given inorder and postorder traversal of a tree, construct the binary tree.

要知道如何构建二叉树，首先我们需要知道二叉树的几种遍历方式，譬如有如下的二叉树：



```
                    1
           --------|-------
           2                   3
       ----|----           ----|----
      4         5         6         7
```

- 前序遍历 1245367
- 中序遍历 4251637
- 后续遍历 4526731

具体到上面这一题，我们知道了一个二叉树的中序遍历以及后序遍历的结果，那么如何构建这颗二叉树呢？

仍然以上面那棵二叉树为例，我们可以发现，对于后序遍历来说，最后一个元素一定是根节点，也就是1。然后我们在中序遍历的结果里面找到1所在的位置，那么它的左半部分就是其左子树，有半部分就是其右子树。

我们将中序遍历左半部分425取出，同时发现后序遍历的结果也在相应的位置上面，只是顺序稍微不一样，也就是452。我们可以发现，后序遍历中的2就是该子树的根节点。

上面说到了左子树，对于右子树，我们取出637，同时发现后序遍历中对应的数据偏移了一格，并且顺序也不一样，为673。而3就是这颗右子树的根节点。

重复上述过程，通过后续遍历找到根节点，然后在中序遍历数据中根据根节点拆分成两个部分，同时将对应的后序遍历的数据也拆分成两个部分，重复递归，就可以得到整个二叉树了。

代码如下：

```
class Solution {
public:
    unordered_map<int, int> m;
    TreeNode *buildTree(vector<int> &inorder, vector<int>
&postorder) {
        if(postorder.empty()) {
            return NULL;
        }

        for(int i = 0; i < inorder.size(); i++) {
            m[inorder[i]] = i;
        }

        return build(inorder, 0, inorder.size() - 1,
            postorder, 0, postorder.size() - 1);
    }

    TreeNode* build(vector<int>& inorder, int s0, int e0,
        vector<int>& postorder, int s1, int e1) {
        if(s0 > e0 || s1 > e1) {
            return NULL;
        }

        TreeNode* root = new TreeNode(postorder[e1]);

        int mid = m[postorder[e1]];
        int num = mid - s0;

        root->left = build(inorder, s0, mid - 1,
postorder, s1, s1 + num - 1);
        root->right = build(inorder, mid + 1, e0,
postorder, s1 + num, e1 - 1);

        return root;
    }
```

```
};
```

这里我们需要注意，为了保证快速的在中序遍历结果里面找到根节点，我们使用了 hash map。

Construct Binary Tree from Preorder and Inorder Traversal

> Given preorder and inorder traversal of a tree, construct the binary tree.

这题跟上面那题类似，通过前序遍历和中序遍历的结果构造二叉树，我们只需要知道前序遍历的第一个值就是根节点，那么仍然可以采用上面提到的方式处理：

- 通过前序遍历找到根节点
- 通过根节点将中序遍历数据拆分成两部分
- 对于各个部分重复上述操作

代码如下：

```
class Solution {
public:
     unordered_map<int, int> m;
    TreeNode *buildTree(vector<int> &preorder,
vector<int> &inorder) {
        if(preorder.empty()) {
            return NULL;
        }

        for(int i = 0; i < inorder.size(); i++) {
            m[inorder[i]] = i;
        }

        return build(preorder, 0, preorder.size() - 1,
inorder, 0, inorder.size() - 1);
    }
```

```
    TreeNode* build(vector<int>& preorder, int s0, int
e0, vector<int> &inorder, int s1, int e1) {
        if(s0 > e0 || s1 > e1) {
            return NULL;
        }

        int mid = m[preorder[s0]];

        TreeNode* root = new TreeNode(preorder[s0]);

        int num = mid - s1;

        root->left = build(preorder, s0 + 1, s0 + num,
inorder, s1, mid - 1);
        root->right = build(preorder, s0 + num + 1, e0,
inorder, mid + 1, e1);

        return root;
    }
};
```

可以看到，这两道题目，只要能清楚了树的几种遍历方式，以及找到如何找到根节点，并通过中序遍历拆分成两个子树，就能很容易的搞定了，唯一需要注意的是写代码的时候拆分索引位置要弄对。

Binary Tree Level Order Traversal

> Given a binary tree, return the level order traversal of its nodes' values. (ie, from left to right, level by level).
>
> For example: Given binary tree {3,9,20,#,#,15,7},

```
    3
   / \
  9  20
    /  \
   15   7
```

> return its level order traversal as:

```
[
  [3],
  [9,20],
  [15,7]
]
```

题目翻译: 给定一颗二叉树，返回一个二维数组，使这个二维数组的每一个元素代表着二叉树的一层的元素.例子已经明确给出.

题目分析: 对于二叉树的问题，我们第一想到的就是DFS或者BFS, DFS更易于理解代码，如果处理数据量不是很大的话.对于这样的面试题，我建议用DFS来求解.

需要注意的点为:

1. 对于一棵树，如果我们要求每一层的节点，并且存在一个二维数组里，首先我们要建一个二维数组，但是这个二维数组建多大的合适呢？我们就要求出这颗树的深度，根据深度来建立二维数组.
2. 题目要求为从左往右添加,所以我们也就是要先放左边的节点，再放右边的节点.
3. 对于这道题，我们首先就是要用DFS来求出这颗树的高度，之后再用DFS对于

每一层遍历，这样节省了空间复杂度.

时间复杂度分析: 由于两次DFS是并列的，并没有嵌套，所以我们的时间复杂度为O(n).

代码如下:

```
/**
 * Definition for binary tree
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL)
{}
 * };
 */
class Solution {
public:
/* for this question, we need to construct the ret vector
first
   thus, we need to know the depth of this tree, we write
a simple
   function to calculate the height of this tree */
    vector<vector<int> > levelOrder(TreeNode *root) {
        int depth = getHeight(root);
        vector<vector<int>> ret(depth);
        if(depth == 0) //invalid check
             return ret;
         getSolution(ret,root,0);
         return ret;
    }

    void getSolution(vector<vector<int>>& ret, TreeNode*
root, int level)
    {
         if(root == NULL)
```

```
            return;
        ret[level].push_back(root->val);
        getSolution(ret,root->left,level+1);
        getSolution(ret,root->right,level+1);
    }

    int getHeight(TreeNode* root)
    {
        if(root == NULL)
            return 0;
        int left = getHeight(root->left);
        int right = getHeight(root->right);
        int height = (left > right?left:right)+1;
        return height;
    }
};
```

Binary Tree Level Order Traversal II

> Given a binary tree, return the bottom-up level order traversal of its nodes' values. (from left to right, level by level from leaf to root)
>
> For example: Given binary tree {3,9,20,#,#,15,7},

```
    3
   / \
  9  20
    /  \
   15   7
```

> return its level order traversal as:

```
[
  [15,7],
  [9,20],
  [3]
]
```

题目翻译: 给定一颗二叉树， 返回一个二维数组，这个二维数组要满足这个条件，二维数组的第一个一维数组要是这可二叉树的最下面一层，之后以此类推，根据以上例子应该知道要求的条件。

题目分析 && 解题思路: 这道题和Binary Tree Level Order Traversal 几乎是一摸一样的，唯一不同的就是二维数组的存储顺序，详见以下代码.

时间复杂度: O(n)-树的dfs均为O(n)

代码如下:

```
/**
 * Definition for binary tree
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL)
{}
 * };
 */
class Solution {
public:
    vector<vector<int> > levelOrderBottom(TreeNode *root)
{
        int depth = height(root);
        vector<vector<int>> ret(depth);
        if(depth == 0)
            return ret;
```

```
        DFS(ret,ret.size()-1, root);
        return ret;
    }

    void DFS(vector<vector<int>>& ret, int level,
TreeNode* root)
    {
        if(root == NULL)
            return;
        ret[level].push_back(root->val);
        DFS(ret,level-1,root->left);
        DFS(ret,level-1,root->right);
    }

    /* get the height first of all */
    int height(TreeNode* root)
    {
        if(root == NULL)
            return 0;
        int left_side = height(root->left);
        int right_side = height(root->right);
        int height = (left_side > right_side?
left_side:right_side)+1;
        return height;
    }
};
```

Binary Tree Zigzag Level Order Traversal

> Given a binary tree, return the zigzag level order traversal of its nodes' values. (ie, from left to right, then right to left for the next level and alternate between).
>
> For example: Given binary tree {3,9,20,#,#,15,7},
>
> 
>
> ```
>     3
>    / \
>   9  20
>     /  \
>    15   7
> ```
>
> return its zigzag level order traversal as:
>
> ```
> [
>   [3],
>   [20,9],
>   [15,7]
> ]
> ```

如果完成了上面两题，这题应该是很简单的，我们只需要将得到的数据按照zigzag的方式翻转一下，代码如下：

```
class Solution {
public:
vector<vector<int> > vals;
    vector<vector<int> > zigzagLevelOrder(TreeNode *root)
{
        build(root, 1);

        //翻转
        for(int i = 1; i < vals.size(); i+=2) {
            reverse(vals[i].begin(), vals[i].end());
        }

```

```
        return vals;
    }

    void build(TreeNode* node, int level) {
        if(!node) {
            return;
        }

        if(vals.size() <= level - 1) {
            vals.push_back(vector<int>());
        }

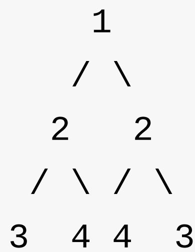
        vals[level - 1].push_back(node->val);

        if(node->left) {
            build(node->left, level + 1);
        }

        if(node->right) {
            build(node->right, level + 1);
        }
    }
};
```

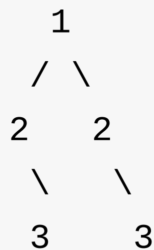
Symmetric Tree

> Given a binary tree, check whether it is a mirror of itself(ie, symmetric around its center)
>
> For example, this tree is symmetric:



```
    1
   / \
  2   2
 / \ / \
3  4 4  3
```

> But the following tree is not.



```
    1
   / \
  2   2
   \   \
   3    3
```

题目翻译： 判断一棵树是不是自己的镜像， 根据以上正反两个例子，我想大家都明白这道题的题目要求了，简单明了.

解题思路: 递归: 这道题没什么特别的地方，现在这里简单的分析一下解题思路，从根节点往下，我们要判断三个条件.

1. 左右两个节点的大小是否相同.
2. 左节点的左孩子是否和右节点的右孩子相同.
3. 左节点的右孩子是否和右节点的左孩子相同.

循环: 这道题的难点在于循环解法， 如果是循环解法，我们必须要用到额外的存储空间用于回溯，关键是对于这道题目， 我们要用多少？要怎么用？要用什么样的存储空间？

递归求解，如果以上三个条件对于每一层都满足，我们就可以认为这棵树是镜像树.

时间复杂度: 递归:本质其实就是DFS,时间复杂度为O(n),空间复杂度O(1) 递归:时间复杂度O(n),空间复杂度O(n)

递归代码如下:

```
/**
 * Definition for binary tree
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL)
{}
 * };
 */
class Solution {
public:
    bool isSymmetric(TreeNode *root) {
        if(root == NULL)
            return true;
        return Helper(root->left,root->right);
    }

    bool Helper(TreeNode* left, TreeNode* right)
    {
        if(left == NULL&&right == NULL)
            return true;
        else if(left == NULL||right == NULL)
            return false;
        bool cond1 = left->val == right->val;
        bool cond2 = Helper(left->left,right->right);
        bool cond3 = Helper(left->right, right->left);
        return cond1&&cond2&&cond3;
    }

};
```

循环解法: 我们主要想介绍一下这道题的循环解法，对于循环，我们要满足对于每一层进行check，代替了递归，一般树的循环遍历，我们都是用FIFO的queue来作为临时空间存储变量的，所以这道题我们也选取了queue，但是我们用两个queue，因为我们要对于左右同时进行检查，很显然一个queue是不够的，具体实现细节，咱们还是看代码吧，，我相信代码更能解释方法.

循环代码如下:

```
/**
 * Definition for binary tree
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL)
{}
 * };
 */
class Solution {
public:
    bool isSymmetric(TreeNode *root) {
       if(root == NULL)
            return true;
        TreeNode* n1 = root->left;
        TreeNode* n2 = root->right;
        if(!n1&&!n2)
            return true;
        if((!n1&&n2)||(n1&&!n2))
            return false;
        queue<TreeNode*> Q1;
        queue<TreeNode*> Q2;
        Q1.push(n1);
        Q2.push(n2);
        while(!Q1.empty() && !Q2.empty())
        {
```

```
            TreeNode* tmp1 = Q1.front();
            TreeNode* tmp2 = Q2.front();
            Q1.pop();
            Q2.pop();
            if((!tmp1&&tmp2) || (tmp1&&!tmp2))
                return false;
            if(tmp1&&tmp2)
            {
                if(tmp1->val != tmp2->val)
                    return false;
                Q1.push(tmp1->left);
                Q1.push(tmp1->right); //note: this line
we should put the mirror sequence in two queues
                Q2.push(tmp2->right);
                Q2.push(tmp2->left);
            }
        }
        return true;
    }
};
```

Same Tree

> Given two binary trees, write a function to check if they are equal or not. Two binary trees are considered equal if they are structurally identical and the nodes have the same values.

题目翻译: 给两棵树，写一个函数来判断这两棵树是否相同. 我们判定一棵树是否相同的条件为这两棵树的结构相同，并且每个节点的值相同.

解题思路: 这道题中规中矩，很简单，我们直接用DFS前序遍历这两棵树就可以了.

时间复杂度分析: 因为是DFS, 所以时间复杂度为O(n)

代码如下:

```
/**
 * Definition for binary tree
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL)
{}
 * };
 */
class Solution {
public:
    bool isSameTree(TreeNode *p, TreeNode *q) {
        if(p == NULL && q == NULL)
            return true;
        else if(p == NULL || q == NULL)
            return false;
        if(p->val == q->val)
        {
            bool left = isSameTree(p->left, q->left);
            bool right = isSameTree(p->right,q->right);
            return left&&right;
        }
        return false;
    }
};
```

Balanced Binary Tree

> Given a binary tree, determine if it is height-balanced.
>
> For this problem, a height-balanced binary tree is defined as a binary tree in which the depth of the two subtrees of every node never differ by more than 1.

题目翻译: 给定一颗二叉树， 写一个函数来检测这棵树是否是平衡二叉树. 对于这个问题, 一颗平衡树的定义是其中任意节点的左右子树的高度差不大于1.

解题思路: 这道题其实就是应用DFS，对于一颗二叉树边计算树的高度边计算差值，针对树里面的每一个节点计算它的左右子树的高度差，如果差值大于1，那么就返回-1，如果不大于1，从下往上再次检测.

时间复杂度: 由于是运用DFS，所以时间复杂度为O(n).

代码如下:

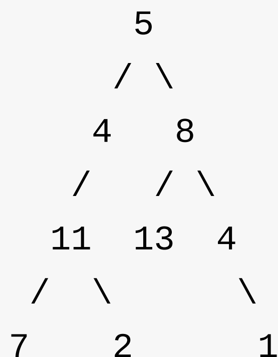
```
/**
 * Definition for binary tree
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL)
{}
 * };
 */
class Solution {
public:
    bool isBalanced(TreeNode *root) {
        //corner case check
        if(root == NULL)
            return true;
```

```
        int isBalanced = getHeight(root);
        if(isBalanced != -1)
            return true;
        else
            return false;
    }

    int getHeight(TreeNode* root)
    {
        if(root == NULL)
            return 0;
        int leftHeight = getHeight(root->left);
        if(leftHeight == -1)
            return -1;
        int rightHeight = getHeight(root->right);
        if(rightHeight == -1)
            return -1;
        int diffHeight = rightHeight > leftHeight?
rightHeight-leftHeight:leftHeight-rightHeight;
        if(diffHeight > 1)
            return -1;
        else
            return diffHeight = (rightHeight>leftHeight?
rightHeight:leftHeight)+1;
    }
};
```

Path Sum

> Given a binary tree and a sum, determine if the tree has a root-to-leaf path such that adding up all the values along the path equals the given sum.
>
> For example: Given the below binary tree and sum = 22,



```
          5
         / \
        4   8
       /   / \
      11  13  4
     /  \      \
    7    2      1
```

> return true, as there exist a root-to-leaf path 5->4->11->2 which sum is 22.

题目翻译: 给定一颗二叉树和一个特定值，写一个方法来判定这棵树是否存在这样一种条件，使得从root到其中一个叶子节点的路径的和等于给定的sum值.

解题思路: 这道题很常规，直接用DFS就可以求解.

时间复杂度: O(n)

代码如下:

```
/**
 * Definition for binary tree
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL)
{}
 * };
 */
```

```
class Solution {
public:
    bool hasPathSum(TreeNode *root, int sum) {
        if(root == NULL)
            return false;
        return DFS(sum, 0, root);
    }

    bool DFS(int target, int sum, TreeNode* root)
    {
        if(root == NULL)
            return false;
        sum += root->val;
        if(root->left == NULL && root->right == NULL)
        {
            if(sum == target)
                return true;
            else
                return false;
        }
        bool leftPart = DFS(target, sum, root->left);
        bool rightPart = DFS(target, sum, root->right);
        return leftPart||rightPart;
    }

};
```

Binary Tree Depth Order Traversal

前面我们解决了tree的level order遍历问题，这里我们需要来处理tree的depth order，也就是前序，中序和后序遍历。

Binary Tree Preorder Traversal

> Given a binary tree, return the preorder traversal of its nodes' values.
>
> For example: Given binary tree {1,#,2,3},
>
> ```
>     1
>      \
>       2
>      /
>     3
> ```
>
> return [1,2,3].
>
> Note: Recursive solution is trivial, could you do it iteratively?

给定一颗二叉树，使用迭代的方式进行前序遍历。

因为不能递归，所以我们只能使用stack来保存迭代状态。

对于前序遍历，根节点是最先访问的，然后是左子树，最后才是右子树。当访问到根节点的时候，我们需要将右子树压栈，这样访问左子树之后，才能正确地找到对应的右子树。

代码如下：

```
class Solution {
public:
    vector<int> preorderTraversal(TreeNode *root) {
        vector<int> vals;
        if(root == NULL) {
            return vals;
        }

        vector<TreeNode*> nodes;

        //首先将root压栈
        nodes.push_back(root);

        while(!nodes.empty()) {
            TreeNode* n = nodes.back();
            vals.push_back(n->val);

            //访问了该节点，出栈
            nodes.pop_back();

            //如果有右子树，压栈保存
            if(n->right) {
                nodes.push_back(n->right);
            }

            //如果有左子树，压栈保存
            if(n->left) {
                nodes.push_back(n->left);
            }
        }

        return vals;
    }
};
```

Binary Tree Inorder Traversal

给定一颗二叉树，使用迭代的方式进行中序遍历。

对于中序遍历，首先遍历左子树，然后是根节点，最后才是右子树，所以我们需要用stack记录每次遍历的根节点，当左子树遍历完成之后，从stack弹出根节点，得到其右子树，开始新的遍历。

代码如下：

```
class Solution {
public:
    vector<int> inorderTraversal(TreeNode *root) {
        vector<int> vals;
        if(root == NULL) {
            return vals;
        }

        vector<TreeNode*> nodes;
        TreeNode* p = root;
        while(p || !nodes.empty()) {
            //这里一直遍历左子树，将根节点压栈
            while(p) {
                nodes.push_back(p);
                p = p->left;
            }

            if(!nodes.empty()) {
                p = nodes.back();
                vals.push_back(p->val);

                //将根节点弹出，获取右子树
                nodes.pop_back();
                p = p->right;
            }
        }

        return vals;
    }
};
```

Binary Tree Postorder Traversal

给定一颗二叉树，使用迭代的方式进行后序遍历。

对于后序遍历，首先遍历左子树，然后是右子树，最后才是根节点。当遍历到一个节点的时候，首先我们将右子树压栈，然后将左子树压栈。这里需要注意一下出栈的规则，对于叶子节点来说，直接可以出栈，但是对于根节点来说，我们需要一个变量记录上一次出栈的节点，如果上一次出栈的节点是该根节点的左子树或者右子树，那么该根节点可以出栈，否则这个根节点是新访问的节点，将右和左子树分别压栈。

代码如下：

```
class Solution {
public:
    vector<int> postorderTraversal(TreeNode *root) {
        vector<int> vals;
        if(root == NULL) {
            return vals;
        }

        vector<TreeNode*> nodes;
        TreeNode* pre = NULL;

        nodes.push_back(root);

        while(!nodes.empty()) {
            TreeNode* p = nodes.back();
            //如果不判断pre，我们就没法正确地出栈了
            if((p->left == NULL && p->right == NULL) ||
                (pre != NULL && (pre == p->left || pre ==
p->right))) {
                vals.push_back(p->val);
                nodes.pop_back();
                pre = p;
            } else {
                //右子树压栈
                if(p->right != NULL) {
                    nodes.push_back(p->right);
                }
```

```
                //左子树压栈
                if(p->left != NULL) {
                    nodes.push_back(p->left);
                }
            }
        }

        return vals;
    }
};
```

总结

可以看到，树的遍历通过递归或者堆栈的方式都是比较容易的，网上还有更牛的不用栈的方法，只是我没理解，就不做过多说明了。

Populating Next Right Pointers in Each Node

> Given a binary tree
>
> ```
>     struct TreeLinkNode {
>       TreeLinkNode *left;
>       TreeLinkNode *right;
>       TreeLinkNode *next;
>     }
> ```
>
> Populate each next pointer to point to its next right node. If there is no next right node, the next pointer should be set to NULL.
>
> Initially, all next pointers are set to NULL.
>
> Note:
>
> You may only use constant extra space. You may assume that it is a perfect binary tree (ie, all leaves are at the same level, and every parent has two children). For example, Given the following perfect binary tree,
>
> ```
>          1
>        /  \
>       2    3
>      / \  / \
>     4  5  6  7
> ```
>
> After calling your function, the tree should look like:
>
> ```
>          1 -> NULL
>        /  \
>       2 -> 3 -> NULL
>      / \  / \
>     4->5->6->7 -> NULL
> ```

这题需要在一棵完全二叉树中使用next指针连接旁边的节点，我们可以发现一些规律。

- 如果一个子节点是根节点的左子树，那么它的next就是该根节点的右子树，譬

如上面例子中的4，它的next就是2的右子树5。

- 如果一个子节点是根节点的右子树，那么它的next就是该根节点next节点的左子树。譬如上面的5，它的next就是2的next（也就是3）的左子树。

所以代码如下：

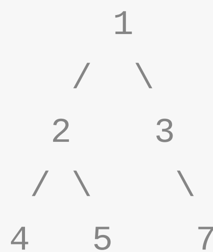
```
class Solution {
public:
    void connect(TreeLinkNode *root) {
        if(!root) {
            return;
        }

        TreeLinkNode* p = root;
        TreeLinkNode* first = NULL;
        while(p) {
            //记录下层第一个左子树
            if(!first) {
                first = p->left;
            }
            //如果有左子树，那么next就是父节点
            if(p->left) {
                p->left->next = p->right;
            } else {
                //叶子节点了，遍历结束
                break;
            }

            //如果有next，那么设置右子树的next
            if(p->next) {
                p->right->next = p->next->left;
                p = p->next;
                continue;
            } else {
                //转到下一层
                p = first;
```

```
                    first = NULL;
                }
            }
        }
};
```

Populating Next Right Pointers in Each Node II

> What if the given tree could be any binary tree? Would your previous solution still work?
>
> Note:
>
> You may only use constant extra space. For example, Given the following binary tree,
>
> 
>
> ```
>          1
>        /  \
>       2    3
>      / \    \
>     4   5    7
> ```
>
> After calling your function, the tree should look like:
>
> 
>
> ```
>          1 -> NULL
>        /  \
>       2 -> 3 -> NULL
>      / \    \
>     4-> 5 -> 7 -> NULL
> ```

不同于上一题，这题的二叉树并不是完全二叉树，我们不光需要提供first指针用来表示一层的第一个元素，同时也需要使用另一个lst指针表示该层上一次遍历的元素。那么我们只需要处理好如何设置last的next指针就可以了。

代码如下:

```
class Solution {
public:
    void connect(TreeLinkNode *root) {
        if(!root) {
            return;
        }

        TreeLinkNode* p = root;
        TreeLinkNode* first = NULL;
        TreeLinkNode* last = NULL;

        while(p) {
            //设置下层第一个元素
            if(!first) {
                if(p->left) {
                    first = p->left;
                } else if(p->right) {
                    first = p->right;
                }
            }

            if(p->left) {
                //如果有last，则设置last的next
                if(last) {
                    last->next = p->left;
                }
                //last为left
                last = p->left;
            }

            if(p->right) {
                //如果有last，则设置last的next
                if(last) {
                    last->next = p->right;
```

```
                }
                //last为right
                last = p->right;
            }

            //如果有next，则转到next
            if(p->next) {
                p = p->next;
            } else {
                //转到下一层
                p = first;
                last = NULL;
                first = NULL;
            }
        }
    }
};
```

其实我们可以看到，第一题只是第二题的特例，所以第二题的解法也同样适用于第一题。

Convert Sorted List to Binary Search Tree

> Given a singly linked list where elements are sorted in ascending order, convert it to a height balanced BST.

这题需要将一个排好序的链表转成一个平衡二叉树，我们知道，对于一个二叉树来说，左子树一定小于根节点，而右子树大于根节点。所以我们需要找到链表的中间节点，这个就是根节点，链表的左半部分就是左子树，而右半部分则是右子树，我们继续递归处理相应的左右部分，就能够构造出对应的二叉树了。

这题的难点在于如何找到链表的中间节点，我们可以通过fast，slow指针来解决，fast每次走两步，slow每次走一步，fast走到结尾，那么slow就是中间节点了。

代码如下：

```
class Solution {
public:

    TreeNode *sortedListToBST(ListNode *head) {
        return build(head, NULL);
    }

    TreeNode* build(ListNode* start, ListNode* end) {
        if(start == end) {
            return NULL;
        }

        ListNode* fast = start;
        ListNode* slow = start;

        while(fast != end && fast->next != end) {
            slow = slow->next;
            fast = fast->next->next;
        }

        TreeNode* node = new TreeNode(slow->val);
        node->left = build(start, slow);
        node->right = build(slow->next, end);

        return node;
    }
};
```

Convert Sorted Array to Binary Search Tree

> Given an array where elements are sorted in ascending order, convert it to a height balanced BST.

这题类似上面那题，同样地解题方式，对于数组来说，能更方便的得到中间节点，代码如下：

```
class Solution {
public:
    TreeNode *sortedArrayToBST(vector<int> &num) {
        return build(num, 0, num.size());
    }

    TreeNode* build(vector<int>& num, int start, int end)
{
        if(start >= end) {
            return NULL;
        }

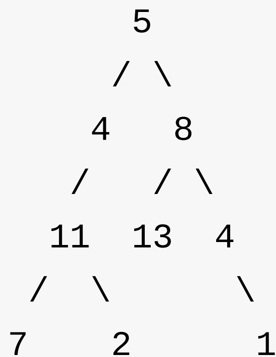
        int mid = start + (end - start) / 2;

        TreeNode* node = new TreeNode(num[mid]);
        node->left = build(num, start, mid);
        node->right = build(num, mid + 1, end);

        return node;
    }
};
```

Path Sum II

> Given a binary tree and a sum, find all root-to-leaf paths where each path's sum equals the given sum.
>
> For example: Given the below binary tree and sum = 22.



```
              5
             / \
            4   8
           /   / \
          11  13  4
         /  \      \
        7    2      1
```

> return

```
[
   [5,4,11,2],
   [5,8,4,5]
]
```

题目翻译： 给定一个二叉树，并且给定一个值，找出所有从根节点到叶子节点和等于这个给定值的路径.上面的例子可以很好地让读者理解这个题目的目的.

解题思路: 这个题目和Path Sum的解法几乎是一模一样，都是用dfs来进行求解，不过就是在传参数的时候有些不同了，因为题目的要求也不同.

时间复杂度: O(n)

代码如下:

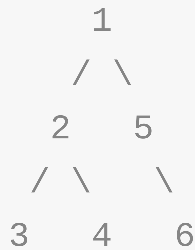
```
/**
 * Definition for binary tree
```

```
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL)
{}
 * };
 */
class Solution {
public:
    vector<vector<int> > pathSum(TreeNode *root, int sum)
{
        vector<vector<int>> ret;
        if(root == NULL)
            return ret;
        vector<int> curr;
        DFS(ret,curr,sum,0,root);
        return ret;
    }

    void DFS(vector<vector<int>>& ret, vector<int> curr,
int sum, int tmpsum, TreeNode* root)
    {
        if(root == NULL)
            return;
        tmpsum+=root->val;
        curr.push_back(root->val);
        if(tmpsum == sum)
        {
            if(root->left == NULL&&root->right == NULL)
            {
                ret.push_back(curr);
                return;
            }
        }
        DFS(ret,curr,sum,tmpsum,root->left);
```

```
            DFS(ret,curr,sum,tmpsum,root->right);
        }
    };
```

Flatten Binary Tree to Linked List

> Given a binary tree, flatten it to a linked list in-place.
>
> For example, Given
>
> 
>
> The flattened tree should look like:
>
> 

给定一颗二叉树，将其扁平化处理，我们可以看到处理之后的节点顺序其实跟前序遍历原二叉树的一致，所以我们只需要前序遍历二叉树同时处理就可以了。代码如下：

```
class Solution {
public:
    void flatten(TreeNode *root) {
                if(!root) {
```

```
            return;
        }

        vector<TreeNode*> ns;
        TreeNode dummy(0);

        TreeNode* n = &dummy;

        ns.push_back(root);

        while(!ns.empty()) {
            TreeNode* p = ns.back();
            ns.pop_back();

            //挂载到右子树
            n->right = p;
            n = p;


            //右子树压栈
            if(p->right) {
                ns.push_back(p->right);
                p->right = NULL;
            }

            //左子树压栈
            if(p->left) {
                ns.push_back(p->left);
                p->left = NULL;
            }
        }
    }
};
```

Validate Binary Search Tree

> Given a binary tree, determine if it is a valid binary search tree (BST).
>
> Assume a BST is defined as follows:
>
> - The left subtree of a node contains only nodes with keys less than the node's key.
> - The right subtree of a node contains only nodes with keys greater than the node's key.
> - Both the left and right subtrees must also be binary search trees.

这题需要判断是不是一个正确的二叉搜索树，比较简单地一道题。

我们通过递归整棵树来解决，代码如下：

```
class Solution {
public:
    bool isValidBST(TreeNode *root) {
        return valid(root, numeric_limits<int>::min(),
numeric_limits<int>::max());
    }

    bool valid(TreeNode* node, int minVal, int maxVal) {
        if(!node) {
            return true;
        }

        if(node->val <= minVal || node->val >= maxVal) {
            return false;
        }

        return valid(node->left, minVal, node->val) &&
        valid(node->right, node->val, maxVal);
    }
};
```

Recover Binary Search Tree

> Two elements of a binary search tree (BST) are swapped by mistake.
>
> Recover the tree without changing its structure.
>
> Note:
>
> A solution using O(n) space is pretty straight forward. Could you devise a constant space solution?

这题需要修复一颗二叉搜索树的两个交换节点数据，我们知道对于一颗二叉搜索树来说，如果按照中序遍历，那么它输出的值是递增有序的，所以我们只需要按照中序遍历输出，在输出结果里面找到两个异常数据（比它后面输出结果大），交换这两个节点的数据就可以了。

但是这题要求使用O(1)的空间，如果采用通常的中序遍历（递归或者栈）的方式，都需要O(N)的空间，所以这里我们用Morris Traversal的方式来进行树的中序遍历。

Morris Traversal中序遍历的原理比较简单：

- 如果当前节点的左孩子为空，则输出当前节点并将其有孩子作为当前节点。
- 如果当前节点的左孩子不为空，在当前节点的左子树中找到当前节点在中序遍历下的前驱节点，也就是当前节点左子树的最右边的那个节点。
    - 如果前驱节点的右孩子为空，则将它的右孩子设置为当前节点，当前节点更新为其左孩子。
    - 如果前驱节点的右孩子为当前节点，则将前驱节点的右孩子设为空，输出当前节点，当前节点更新为其右孩子。

重复上述过程，直到当前节点为空，递归的时候我们同时需要记录错误的节点。那么我们如何知道一个节点的数据是不是有问题呢？对于中序遍历来说，假设当前节点为cur，它的前驱节点为pre，如果cur的值小于pre的值，那么cur和pre里面的数据就是交换的了。

代码如下：

```
class Solution {
public:
```

```
    void recoverTree(TreeNode *root) {
        TreeNode* cur = 0;
        TreeNode* pre = 0;
        TreeNode* p1 = 0;
        TreeNode* p2 = 0;
        TreeNode* preCur = 0;

        bool found = false;

        if(!root) {
            return;
        }

        cur = root;
        while(cur) {
            if(!cur->left) {
                //记录p1和p2
                if(preCur && preCur->val > cur->val) {
                    if(!found) {
                        p1 = preCur;
                        found = true;
                    }
                    p2 = cur;
                }

                preCur = cur;
                cur = cur->right;
            } else {
                pre = cur->left;
                while(pre->right && pre->right != cur) {
                    pre = pre->right;
                }

                if(!pre->right) {
                    pre->right = cur;
                    cur = cur->left;
```

```
                } else {
                    //记录p1和p2
                    if(preCur->val > cur->val) {
                        if(!found) {
                            p1 = preCur;
                            found = true;
                        }
                        p2 = cur;
                    }
                    preCur = cur;
                    pre->right = NULL;
                    cur = cur->right;
                }
            }
        }

        if(p1 && p2) {
            int t = p1->val;
            p1->val = p2->val;
            p2->val = t;
        }
    }
};
```

Binary Tree Path

> Given a binary tree, return all root-to-leaf paths.
>
> For example, given the following binary tree:
>
> 
>
> ```
>    1
>  /   \
> 2     3
>  \
>   5
> ```
>
> All root-to-leaf paths are:
>
> ```
> ["1->2->5", "1->3"]
> ```

题目翻译：给定一棵二叉树，返回所有从根节点到叶节点的路径。

题目分析：本题属于二叉树的遍历问题，可以用深度优先搜索来解决。

使用栈来记录遍历过程中访问过的节点。递归地访问每个节点的子节点，如果遇到叶节点，则输出记录的路径。返回上一层之前弹出栈顶元素。C++的vector容器也能做到后进先出，所以下面的代码并没有使用std::stack类来实现。

生成输出的字符串时，可以使用std::stringstream类来完成，类似于Java和C#中的StringBuilder。

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL)
{}
```

```
 * };
 */
class Solution {
public:
    vector<string> binaryTreePaths(TreeNode* root) {
        vector<string> result;
        if (root == nullptr) return result;
        vector<int> path;
        bfs(root, path, result);
        return result;
    }

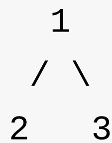
private:
    // 递归函数，深度优先搜索
    void bfs(TreeNode* node, vector<int>& path,
vector<string>& result) {
        if (node == nullptr) return;
        path.push_back(node->val);
        if (node->left == nullptr && node->right ==
nullptr)
            result.push_back(generatePath(path));
        else {
            if (node->left != nullptr) {
                bfs(node->left, path, result);
                path.pop_back();
            }
            if (node->right != nullptr) {
                bfs(node->right, path, result);
                path.pop_back();
            }
        }
    }

    // 辅助函数，用于生成路径字符串
    string generatePath(vector<int> path) {
        stringstream ss;
```

```
        int i;
        for (i = 0; i < path.size() - 1; i++) ss <<
path[i] << "->";
        ss << path[i];
        return ss.str();
    }
};
```

Sum Root to Leaf Numbers

> Given a binary tree containing digits from 0-9 only, each root-to-leaf path could represent a number. An example is the root-to-leaf path `1->2->3` which represents the number `123`. Find the total sum of all root-to-leaf numbers. For example,



```
    1
   / \
  2   3
```

> The root-to-leaf path `1->2` represents the number `12`. The root-to-leaf path `1->3` represents the number `13`. Return the sum = 12 + 13 = 25.

题目翻译： 给定一棵二叉树，仅包含0到9这些数字，每一条从根节点到叶节点的路径表示一个数。例如，路径 `1->2->3` 表示数值123。求出所有路径表示的数值的和。 上述例子中，路径 `1->2` 表示数值12，路径 `1->3` 表示数值13。它们的和是25。

题目分析: 从根节点到叶节点的遍历方法是深度优先搜索(DFS)。解决本题只需在遍历过程中记录路径中的数字，在到达叶节点的时候把记录下来的数字转换成数值，加到和里面即可。

时间复杂度: O(n)

代码如下:

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL)
{}
 * };
```

```
 */
class Solution {
public:
    int sumNumbers(TreeNode* root) {
        vector<int> arr;
        int sum = 0;
        dfs(root, arr, sum);
        return sum;
    }

    int vec2num(vector<int>& vec) {
        int num = 0;
        for (auto n : vec) {
            num = num * 10 + n;
        }
        return num;
    }

    void dfs(TreeNode* node, vector<int>& arr, int& sum)
{
        if (node == nullptr) return;
        arr.push_back(node->val);
        if (node->left == nullptr && node->right ==
nullptr) {
            sum += vec2num(arr);
        } else {
            if (node->left != nullptr) dfs(node->left,
arr, sum);
            if (node->right != nullptr) dfs(node->right,
arr, sum);
        }
        arr.pop_back();
    }

};
```

Dynamic Programming

Best Time to Buy and Sell Stock

> Say you have an array for which the ith element is the price of a given stock on day i.
>
> If you were only permitted to complete at most one transaction (ie, buy one and sell one share of the stock), design an algorithm to find the maximum profit.

这是卖股票的第一个题目，根据题意我们知道只能进行一次交易，但需要获得最大的利润，所以我们需要在最低价买入，最高价卖出，当然买入一定要在卖出之前。

对于这一题，还是比较简单的，我们只需要遍历一次数组，通过一个变量记录当前最低价格，同时算出此次交易利润，并与当前最大值比较就可以了。

代码如下：

```
class Solution {
public:
    int maxProfit(vector<int> &prices) {
        if(prices.size() <= 1) {
            return 0;
        }

        int minP = prices[0];

        int profit = prices[1] - prices[0];

        for(int i = 2; i < prices.size(); i++) {
            minP = min(prices[i - 1], minP);
            profit = max(profit, prices[i] - minP);
        }

        if(profit < 0) {
            return 0;
        }

        return profit;
    }
};
```

Best Time to Buy and Sell Stock II

> Say you have an array for which the ith element is the price of a given stock on day i.
>
> Design an algorithm to find the maximum profit. You may complete as many transactions as you like (ie, buy one and sell one share of the stock multiple times). However, you may not engage in multiple transactions at the same time (ie, you must sell the stock before you buy again).

这题相对于上一题来说更加容易（这不知道为啥是II），因为不限制交易次数，我们在第i天买入，如果发现i + 1天比i高，那么就可以累加到利润里面。

代码如下：

```
class Solution {
public:
    int maxProfit(vector<int> &prices) {
        int len = (int)prices.size();
        if(len <= 1) {
            return 0;
        }

        int sum = 0;
        for(int i = 1; i < len; i++) {
            if(prices[i] - prices[i - 1] > 0) {
                sum += prices[i] - prices[i - 1];
            }
        }

        return sum;
    }
};
```

Best Time to Buy and Sell Stock III

> Say you have an array for which the ith element is the price of a given stock on day i.
>
> Design an algorithm to find the maximum profit. You may complete at most two transactions.
>
> Note: You may not engage in multiple transactions at the same time (ie, you must sell the stock before you buy again).

这题是三道题目中最难的一题，只允许两次股票交易，如果只允许一次，那么题目就退化到第一题了，根据第一题的算法，我们可以得到[0,1,...,i]区间的最大利润，同时在从后往前扫描得到[i,i+1,...,n-1]的最大利润，两个相加就可以得到该题的解了。

代码如下：

```
class Solution {
public:
    int maxProfit(vector<int> &prices) {
        int len = (int)prices.size();
        if(len <= 1) {
            return 0;
        }

        vector<int> profits;
        profits.resize(len);

        //首先我们正向遍历得到每天一次交易的最大收益
        //并保存到profits里面
        int minP = prices[0];
        int sum = numeric_limits<int>::min();
        for(int i = 1; i < len; i++) {
            minP = min(minP, prices[i - 1]);
            profits[i] = max(sum, prices[i] - minP);

            sum = profits[i];
        }

        int maxP = prices[len - 1];
        int sum2 = numeric_limits<int>::min();

        //逆向遍历
        for(int i = len - 2; i >= 0; i--) {
            maxP = max(maxP, prices[i + 1]);
            sum2 = max(sum2, maxP - prices[i]);
```

```
                if(sum2 > 0) {
                    //这里我们直接将其加入profits里面，
                    //不需要额外保存
                    profits[i] = profits[i] + sum2;
                    sum = max(sum, profits[i]);
                }
            }

            return sum > 0 ? sum : 0;
        }
};
```

卖股票的1和3已经涉及到了动态规划，但笔者这方面还未有太多知识积累，所以不能很好的列举出动态规划方程，这是笔者后续需要努力提升的，后续补上。

Unique Paths

> A robot is located at the top-left corner of a m x n grid (marked 'Start' in the diagram below).
>
> The robot can only move either down or right at any point in time. The robot is trying to reach the bottom-right corner of the grid (marked 'Finish' in the diagram below).
>
> How many possible unique paths are there?

这题是一道典型的dp问题，如果机器人要到(i, j)这个点，他可以选择先到(i - 1, j)或者，(i, j - 1)，也就是说，到(i, j)的唯一路径数等于(i - 1, j)加上(i, j - 1)的个数，所以我们很容易得出dp方程:

```
dp[i][j] = dp[i - 1][j] + dp[i][j - 1]
```

dp[i][j]表示从点(0, 0)到(i, j)唯一路径数量。

代码如下：

```
class Solution {
public:
    int uniquePaths(int m, int n) {
        int dp[m][n];
        //初始化dp，m x 1情况全为1
        for(int i = 0; i < m; i++) {
            dp[i][0] = 1;
        }

        //初始化dp，1 x n情况全为1
        for(int j = 0; j < n; j++) {
            dp[0][j] = 1;
        }

        for(int i = 1; i < m; i++) {
            for(int j = 1; j < n; j++) {
                dp[i][j] = dp[i - 1][j] + dp[i][j - 1];
            }
        }

        return dp[m - 1][n - 1];
    }
};
```

Unique Paths II

> Now consider if some obstacles are added to the grids. How many unique paths would there be?
>
> An obstacle and empty space is marked as 1 and 0 respectively in the grid.

这题跟上一题唯一的区别在于多了障碍物，如果某一个点有障碍，那么机器人无法通过。

代码如下:

```
class Solution {
public:
    int uniquePathsWithObstacles(vector<vector<int> >
&obstacleGrid) {
        if(obstacleGrid.empty() ||
obstacleGrid[0].empty()) {
            return 0;
        }

        int m = obstacleGrid.size();
        int n = obstacleGrid[0].size();

        int dp[m][n];

        //下面初始dp的时候需要根据obstacleGrid的值来确定
        dp[0][0] = (obstacleGrid[0][0] == 0 ? 1 : 0);

        //我们需要注意m x 1以及1 x n的初始化
        for(int i = 1; i < m; i++) {
            dp[i][0] = ((dp[i - 1][0] == 1 &&
obstacleGrid[i][0] == 0) ? 1 : 0);
        }

        for(int j = 1; j < n; j++) {
            dp[0][j] = ((dp[0][j - 1] == 1 &&
obstacleGrid[0][j] == 0) ? 1 : 0);
        }


        for(int i = 1; i < m; i++) {
            for(int j = 1; j < n; j++) {
                if(obstacleGrid[i][j] == 1) {
                    dp[i][j] = 0;
                } else {
                    dp[i][j] = dp[i - 1][j] + dp[i][j -
1];
```

```
                }
            }
        }

        return dp[m - 1][n - 1];
    }
};
```

Minimum Path Sum

> Given a m x n grid filled with non-negative numbers, find a path from top left to bottom right which minimizes the sum of all numbers along its path.
>
> Note: You can only move either down or right at any point in time.

这题跟前面两题差不多，所以放到这里说明了。我们使用dp[i][j]表明从(0, 0)到(i, j)最小的路径和，那么dp方程为:

`dp[i][j] = min(dp[i][j-1], dp[i - 1][j]) + grid[i][j]`

代码如下：

```
class Solution {
public:
    int minPathSum(vector<vector<int> > &grid) {
        if(grid.empty() || grid[0].empty()) {
            return 0;
        }

        int row = grid.size();
        int col = grid[0].size();

        int dp[row][col];

        dp[0][0] = grid[0][0];
        for(int i = 1; i < row; i++) {
            dp[i][0] = dp[i - 1][0] + grid[i][0];
        }

        for(int j = 1; j < col; j++) {
            dp[0][j] = dp[0][j - 1] + grid[0][j];
        }

        for(int i = 1; i < row; i++) {
            for(int j = 1; j < col; j++) {
                dp[i][j] = min(dp[i - 1][j], dp[i][j -
1]) + grid[i][j];
            }
        }

        return dp[row - 1][col - 1];
    }
};
```

Maximum Subarray

> Find the contiguous subarray within an array (containing at least one number) which has the largest sum.
>
> For example, given the array [−2,1,−3,4,−1,2,1,−5,4], the contiguous subarray [4,−1,2,1] has the largest sum = 6.

这题是一道经典的dp问题，我们可以很容易的得到其dp方程，假设dp[i]是数组a [0, i]区间最大的值，那么

```
dp[i + 1] = max(dp[i], dp[i] + a[i + 1])
```

代码如下:

```
class Solution {
public:
    int maxSubArray(int A[], int n) {
        int sum = 0;
        int m = INT_MIN;

        for(int i = 0; i < n; i++) {
            sum += A[i];
            m = max(m, sum);
            //如果sum小于0了，将sum重置为0，从i + 1再次开始计算
            if(sum < 0) {
                sum = 0;
            }
        }

        return m;
    }
};
```

虽然这道题目用dp解起来很简单，但是题目说了，问我们能不能采用divide and conquer的方法解答，也就是二分法。

假设数组A[left, right]存在最大区间，mid = (left + right) / 2，那么无非就是三中情况：

1. 最大值在A[left, mid - 1]里面
2. 最大值在A[mid + 1, right]里面
3. 最大值跨过了mid，也就是我们需要计算[left, mid - 1]区间的最大值，以及[mid + 1, right]的最大值，然后加上mid，三者之和就是总的最大值

我们可以看到，对于1和2，我们通过递归可以很方便的求解，然后在同第3的结果比较，就是得到的最大值。

代码如下:

```
class Solution {
public:
    int maxSubArray(int A[], int n) {
        return divide(A, 0, n - 1, INT_MIN);
    }

    int divide(int A[], int left, int right, int tmax) {
        if(left > right) {
            return INT_MIN;
        }

        int mid = left + (right - left) / 2;
        //得到子区间[left, mid - 1]最大值
        int lmax = divide(A, left, mid - 1, tmax);
        //得到子区间[mid + 1, right]最大值
        int rmax = divide(A, mid + 1, right, tmax);

        tmax = max(tmax, lmax);
        tmax = max(tmax, rmax);

        int sum = 0;
        int mlmax = 0;
        //得到[left, mid - 1]最大值
        for(int i = mid - 1; i >= left; i--) {
```

```
                sum += A[i];
                mlmax = max(mlmax, sum);
            }

            sum = 0;
            int mrmax = 0;
            //得到[mid + 1, right]最大值
            for(int i = mid + 1; i <= right; i++) {
                sum += A[i];
                mrmax = max(mrmax, sum);
            }

            tmax = max(tmax, A[mid] + mlmax + mrmax);
            return tmax;
        }
};
```

Maxmimum Product Subarray

> Find the contiguous subarray within an array (containing at least one number) which has the largest product.
>
> For example, given the array [2,3,-2,4], the contiguous subarray [2,3] has the largest product = 6.

这题是求数组中子区间的最大乘积，对于乘法，我们需要注意，负数乘以负数，会变成正数，所以解这题的时候我们需要维护两个变量，当前的最大值，以及最小值，最小值可能为负数，但没准下一步乘以一个负数，当前的最大值就变成最小值，而最小值则变成最大值了。

我们的动态方程可能这样：

```
maxDP[i + 1] = max(maxDP[i] * A[i + 1], A[i + 1],
minDP[i] * A[i + 1])
minDP[i + 1] = min(minDP[i] * A[i + 1], A[i + 1],
maxDP[i] * A[i + 1]
dp[i + 1] = max(dp[i], maxDP[i + 1])
```

这里，我们还需要注意元素为0的情况，如果A[i]为0，那么maxDP和minDP都为0，我们需要从A[i + 1]重新开始。

代码如下:

```
class Solution {
public:
    int maxProduct(int A[], int n) {
        if(n == 0){
            return 0;
        } else if(n == 1) {
            return A[0];
        }

        int p = A[0];
        int maxP = A[0];
        int minP = A[0];
        for(int i = 1; i < n; i++) {
            int t = maxP;
            maxP = max(max(maxP * A[i], A[i]), minP *
A[i]);
            minP = min(min(t * A[i], A[i]), minP * A[i]);
            p = max(maxP, p);
        }

        return p;
    }
};
```

Climbing Stairs

> You are climbing a stair case. It takes n steps to reach to the top.
>
> Each time you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top?

这道题目其实就是斐波那契数列问题，题目比较简单，我们很容易就能列出dp方程

`dp[n] = dp[n - 1] + dp[n - 2]`

初始条件dp[1] = 1, dp[2] = 2。

代码如下：

```
class Solution {
public:
    int climbStairs(int n) {
        int f1 = 2;
        int f2 = 1;
        if(n == 1) {
            return f2;
        } else if(n == 2) {
            return f1;
        }

        int fn;
        for(int i = 3; i <= n; i++) {
            fn = f1 + f2;
            f2 = f1;
            f1 = fn;
        }
        return fn;
    }
};
```

Triangle

> Given a triangle, find the minimum path sum from top to bottom. Each step you may move to adjacent numbers on the row below.
>
> For example, given the following triangle
>
> ```
> [
>      [2],
>     [3,4],
>    [6,5,7],
>   [4,1,8,3]
> ]
> ```
>
> The minimum path sum from top to bottom is 11 (i.e., 2 + 3 + 5 + 1 = 11).

这题要求我们求出一个三角形中从顶到底最小路径和，并且要求只能使用O(n)的空间。

这题有两种解法，自顶向下以及自底向上。

首先来看自顶向下，根据题目我们知道，每向下一层，我们只能选择邻接数字进行累加，譬如上面第1行的数字3，它的下一行邻接数字就是6和5。

我们假设dp[m][n]保存了第m行第n个节点的最小路径和，我们有如下dp方程

- `dp[m + 1][n] = min(dp[m][n], dp[m][n - 1]) + triangle[m + 1][n] if n > 0`
- `dp[m + 1][0] = dp[m][0] + triangle[m + 1][0]`

因为只能使用O(n)的空间，所以我们需要滚动计算，使用一个一位数组保存每层的最小路径和，参考Pascal's Triangle，我们知道，为了防止计算的时候不覆盖以前的值，所以我们需要从后往前计算。

代码如下

```
class Solution {
public:
    int minimumTotal(vector<vector<int> > &triangle) {
        int row = triangle.size();
        if(row == 0) {
            return 0;
        }

        //初始化为最大值
        vector<int> total(row, INT_MAX);
        total[0] = triangle[0][0];
        int minTotal = INT_MAX;
        for(int i = 1; i < row; i++) {
            for(int j = i; j >= 0; j--) {
                if(j == 0) {
                    total[j] = total[j] + triangle[i][j];
                } else {
                    //上一层total[i]为INT_MAX，不会影响最小值
                    total[j] = min(total[j - 1],
total[j]) + triangle[i][j];
                }
            }
        }
        for(int i = 0; i < row; i++) {
            minTotal = min(minTotal, total[i]);
        }
        return minTotal;
    }
};
```

区别于自顶向下，另一种更简单的做法就是自底向上了。dp方程为

`dp[m][n] = min(dp[m + 1][n], dp[m + 1][n + 1]) + triangle[m][n]`

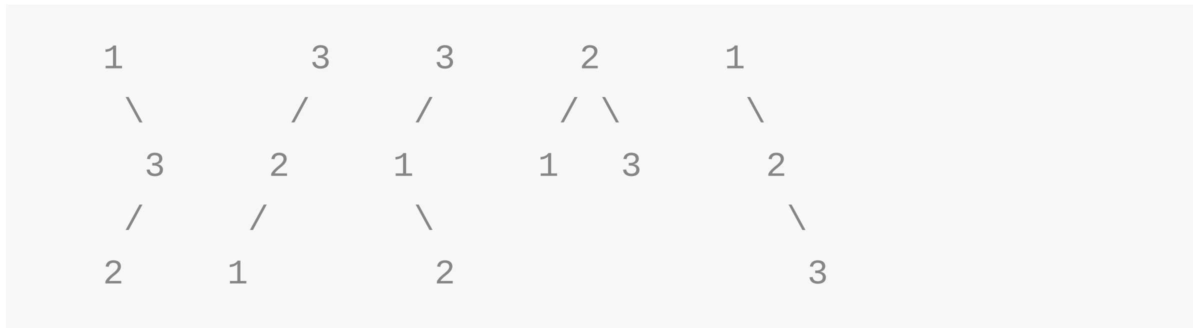
我们仍然可以使用一位数组滚动计算。

代码如下

```
class Solution {
public:
    int minimumTotal(vector<vector<int> > &triangle) {
        if(triangle.empty()) {
            return 0;
        }
        int row = triangle.size();
        vector<int> dp;
        dp.resize(row);
        //用最底层的数据初始化
        for(int i = 0; i < dp.size(); i++) {
            dp[i] = triangle[row-1][i];
        }

        for(int i = row - 2; i >= 0; i--) {
            for(int j = 0; j <= i; j++) {
                dp[j] = triangle[i][j] + min(dp[j], dp[j
+ 1]);
            }
        }
        return dp[0];
    }
};
```

Unique Binary Search Trees

> Given n, how many structurally unique BST's (binary search trees) that store values 1...n?
>
> For example, Given n = 3, there are a total of 5 unique BST's.
>
> 
>
> ```
>    1         3     3      2      1
>     \       /     /      / \      \
>      3     2     1      1   3      2
>     /     /       \                 \
>    2     1         2                 3
> ```

这道题目要求给定一个数n，有多少种二叉树排列方式，用来存储1到n。

刚开始拿到这题的时候，完全不知道如何下手，但考虑到二叉树的性质，对于任意以i为根节点的二叉树，它的左子树的值一定小于i，也就是[0, i - 1]区间，而右子树的值一定大于i，也就是[i + 1, n]。假设左子树有m种排列方式，而右子树有n种，则对于i为根节点的二叉树总的排列方式就是m x n。

我们使用dp[i]表示i个节点下面二叉树的排列个数，那么dp方程为:

```
dp[i] = sum(dp[k] * dp[i - k -1]) 0 <= k < i
```

代码如下：

```
class Solution {
public:
    int numTrees(int n) {
        vector<int> dp(n + 1, 0);

        //dp初始化
        dp[0] = 1;
        dp[1] = 1;

        for(int i = 2; i <= n; i++) {
            for(int j = 0; j < i; j++) {
                //如果左子树的个数为j，那么右子树为i - j - 1
                dp[i] += dp[j] * dp[i - j - 1];
            }
        }

        return dp[n];
    }
};
```

Unique Binary Search Trees II

> Given n, generate all structurally unique BST's (binary search trees) that store values 1...n.
>
> For example, Given n = 3, your program should return all 5 unique BST's shown below.



这题跟前面一题不同，需要得到所有排列的解。

根据前面我们知道，对于在n里面的任意i，它的排列数为左子树[0, i - 1]的排列数 x 右子树[i + 1, n]的排列数，所以我们只需要得到i的左子树和右子树的所有排列，就能得到i的所有排列了。而这个使用递归就能搞定，代码如下：

```
class Solution {
public:
    vector<TreeNode *> generateTrees(int n) {
        return generate(1, n);
    }

    vector<TreeNode*> generate(int start, int stop){
        vector<TreeNode*> vs;
        if(start > stop) {
            //没有子树了，返回null
            vs.push_back(NULL);
            return vs;
        }

        for(int i = start; i <= stop; i++) {
            auto l = generate(start, i - 1);
            auto r = generate(i + 1, stop);

            //获取左子树和右子树所有排列之后，放到root为i的节点
的下面
            for(int j = 0; j < l.size(); j++) {
                for(int k = 0; k < r.size(); k++) {
                    TreeNode* n = new TreeNode(i);
                    n->left = l[j];
                    n->right = r[k];
                    vs.push_back(n);
                }
            }
        }

        return vs;
    }
};
```

Perfect Squares

> Given a positive integer n, find the least number of perfect square numbers (for example, 1, 4, 9, 16, ...) which sum to n. For example, given `n = 12`, return 3 because `12 = 4 + 4 + 4`; given `n = 13`, return 2 because `13 = 4 + 9`.

题目翻译：给出一个正整数 `n` ，求至少需要多少个完全平方数（例如1，4，9，16……）相加能得到n。例如，给出 `n = 12` ，返回3，因为 `12 = 4 + 4 + 4` 。给出 `n = 13` ，返回2，因为 `13 = 4 + 9` 。

题目分析：乍一看题目，比较天真的想法是，先从不大于n的最大的完全平方数开始组合，如果和超过了n，就换小一点的完全平方数。但问题是，最后如果凑不齐的话，只能添加很多1，总量上就不是最少的了。例如12，题目中给的例子是4+4+4，需要3个完全平方数。如果从最大的开始组合，那么是9+1+1+1，需要4个完全平方数。

从另一个角度来想，用穷举法来求解就是把不大于n的所有可能的完全平方数的组合都算出来，然后找出和为n的组合中数量最少的那种组合。如果不大于n的完全平方数有m个的话，这个方法的时间复杂度是O(m^m)。显然穷举法时间复杂度过大，不是可行的方法。观察到，在枚举的过程中，有一些组合显然不是最优的，比如把12拆成12个1相加。另外，如果我们能够记录已经找到的最小组合，那么稍大一些的数只需要在此基础上添加若干个完全平方数即可。这里面就包含了动态规划的思想。

具体来说，我们用一个数组来记录已有的结果，初始化为正无穷(`INT_MAX`)。外层循环变量 `i` 从 `0` 到 `n` ，内层循环变量 `j` 在 `i` 的基础上依次加上每个整数的完全平方，超过 `n` 的不算。那么 `i + j*j` 这个数需要的最少的完全平方数的数量，就是数组中当前的数值，和 `i` 位置的数值加上一，这两者之间较小的数字。如果当前的值较小，说明我们已经找到过它需要的完全平方数的个数（最初都是正无穷）。否则的话，说明在 `i` 的基础上加上 `j` 的平方符合条件，所需的完全平方数的个数就是 `i` 需要的个数加上一。

代码如下

```
class Solution {
public:
    int numSquares(int n) {
        vector<int> dp(n + 1, INT_MAX);
        dp[0] = 0;
        for (int i = 0; i <= n; i++) {
            for (int j = 1; i + j * j <= n; j++) {
                dp[i + j * j] = min(dp[i + j * j], dp[i]
+ 1);
            }
        }
        return dp[n];
    }
};
```

Backtracking

Combination

在这个section里面，我们主要来过一下像leetcode里面类似combination这一系列的题，这类题应该归结为DFS+Backtracking。掌握了大体思想，注意一下边角处理就好，比如剪枝。

先来讨论一下第一题Combination.

Combination

> Given two integers n and k, return all possible combinations of k numbers out of 1,2,...,n.

题目翻译: 给定两个整型数组n和k，要求返由k个数组成的combination，其实应该叫做组合. 这个combination应该是高中里面的组合。这k个数是在n中任选k个数，由题意可得，这里的k应该小于或等于n(这个条件不要忘了做validation check哦).

题目分析: 我觉得应该还有不少读者困惑什么是combination，这里我们先给一个例子比如n=3， k=2的条件下， 所有可能的combination如下： [1,2], [1,3], [2,3]. 注意：[2,3]和[3,2]是相同的，我们只要求有其中一个就可以了. 所以解题的时候，我们要避免相同的组合出现.

解题思路: 看到这道题，首先第一想法应该是用递归来求解.如果要是用循环来求解，这个时间复杂度应该是比较恐怖了.并且，这个递归是一层一层往深处去走的，打个比方，我们一个循环，首先求得以1开始的看个数的combination，之后再求以2开始的，以此类推，所以开始是对n个数做DFS, n-1个数做DFS...所以应该是对n*(n-1)*...*1做DFS. 在程序中，我们可以加一些剪枝条件来减少程序时间.

时间复杂度: 在题目分析中，我们提到了对于对n,n-1,...,1做DFS，所以时间复杂度是O(n!)

代码如下:

```
class Solution {
public:
```

```
    vector<vector<int> > combine(int n, int k) {
        vector<vector<int>> ret;
        if(n <= 0) //corner case invalid check
            return ret;

        vector<int> curr;
        DFS(ret,curr, n, k, 1); //we pass ret as
reference at here
        return ret;
    }

    void DFS(vector<vector<int>>& ret, vector<int> curr,
int n, int k, int level)
    {
        if(curr.size() == k)
        {
            ret.push_back(curr);
            return;
        }
        if(curr.size() > k)  // consider this check to
save run time
            return;

        for(int i = level; i <= n; ++i)
        {
            curr.push_back(i);
            DFS(ret,curr,n,k,i+1);
            curr.pop_back();
        }
    }


};
```

Combination Sum

> Given a set of candidate numbers (C) and a target number (T), find all unique combinations in C where the candidate numbers sums to T.
>
> The same repeated number may be chosen from C unlimited number of times.

Note:

1. All numbers (including target) will be positive integers.
2. Elements in a combination (a1, a2, … , ak) must be in non-descending order. (ie, a1 ≤ a2 ≤ … ≤ ak).
3. The solution set must not contain duplicate combinations.

题目翻译: 给一个数组C和一个目标值T, 找出所有的满足条件的组合：使得组合里面的数字之和等于T,并且一些数字可以从C中午先选择。

注意:

1. 所有给定的数字均为正整数.(这意味着我们加corner case invalid check的时候要加一条，如果给定T不是正整数，我们就没必要在往下进行了)

2. 所有的答案组中要满足升序排列.

3. 最后的答案数组不能包含重复答案.

题目分析: 这道题的大体思路和combination是相同的，不同的地方在于一个数字可以使用多次，这也造成了我们进行实现function的时候要注意的问题，也就是说，传入递归的参数不同于combination.

时间复杂度: 没什么好说的，和combination的时间复杂度是相同的.O(n!)

代码如下:

```
class Solution {
public:
    vector<vector<int> > combinationSum(vector<int>
&candidates, int target) {
        vector<vector<int>> ret;
        //corner case invalid check
```

```
        if(candidates.size() == 0 || target < 0)
            return ret;
        vector<int> curr;
        sort(candidates.begin(),candidates.end());
//because the requirments need the elements should be in
non-descending order
        BackTracking(ret,curr,candidates,target,0);
        return ret;
    }

    /* we use reference at here because the function
return type is 0, make the code understand easily */
    void BackTracking(vector<vector<int>>& ret,
vector<int> curr, vector<int> candidates, int target, int
level)
    {
        if(target == 0)
        {
            ret.push_back(curr);
            return;
        }
        else if(target < 0) //save time
            return;

        for(int i = level; i < candidates.size(); ++i)
        {
            target -= candidates[i];
            curr.push_back(candidates[i]);
            BackTracking(ret,curr,candidates,target,i);
//unlike combination, we do not use i+1 because we can
use the same number multiple times.
            curr.pop_back();
            target += candidates[i];
        }
    }
```

```
};
```

Combination Sum II

> Given a collection of candidate numbers (C) and a target number (T), find all unique combinations in C where the candidate numbers sums to T.
>
> Each number in C may only be used once in the combination.

Note:

1. All numbers (including target) will be positive integers.
2. Elements in a combination (a1, a2, … , ak) must be in non-descending order. (ie, a1 ≤ a2 ≤ … ≤ ak).
3. The solution set must not contain duplicate combinations.

题目翻译: 给定一个数组C和一个特定值T,要求找出这里面满足以下条件的所有答案：数组中数字的值加起来等于特定和的答案.

数组中每个数字只能用一次.（同three sum和four sum的解法）

注意条件:

1. 给定数组的所有值必须是正整数.（意味着我们加corner case invalid check的时候要检查T）
2. 答案数组中的值必须为升序排列.(我们要对数组进行排序)
3. 最终答案不能包含重复数组.

代码如下:

```
class Solution {
public:
    vector<vector<int> > combinationSum2(vector<int>
&num, int target) {
        vector<vector<int>> ret;
        if(num.size() == 0 || target < 0) //invalid
corner case check
```

```
            return ret;
        vector<int> curr;
        sort(num.begin(), num.end());
        BackTracking(ret,curr,num,target,0);
        return ret;
    }

    void BackTracking(vector<vector<int>>& ret,
vector<int> curr, vector<int> num, int target, int level)
    {
        if(target == 0)
        {
            ret.push_back(curr);
            return;
        }
        else if(target < 0)
            return;
        for(int i = level; i < num.size(); ++i)
        {
            target -= num[i];
            curr.push_back(num[i]);
            BackTracking(ret,curr,num,target,i+1);
            curr.pop_back();
            target += num[i];
            while(i < num.size()-1 && num[i] == num[i+1])
//we add this while loop is to skip the duplication
result
                ++i;
        }
    }

};
```

Letter Combinations of a Phone Number

> Given a digit string, return all possible letter combinations that the number could represent. A mapping of digit to letters (just like on the telephone buttons) is given as below: 


```
Input:Digit string "23"
Output: ["ad", "ae", "af", "bd", "be", "bf", "cd", "ce",
"cf"].
```

题目翻译: 给定一个字符串数字，返回这个字符串数字在电话表上可能的combination，一个map的电话键盘在上图已经给出.

题目分析: 这道题目的给出，具体的解题思路是和combination是相同的，不同的地方是我们要先建一个dictionary,以方便查找.之后用combination的相同方法，对于每一个数字，在dictionary中查找它所对应的说有的数字.

解题思路: 我是用字符串数组来构建这个dictionary的，用于下标代表数字，例如，下标为2，我的字典就会有这种对应的关系:dic[2] = "abc".只要把给定数字字符串的每一个数字转换为int类型，就可以根据字典查找出这个数字所对应的所有字母.当然，再构建字典的时候，我们需要注意dic[0] = "", dic[1] = "".这两个特殊的case，因为电话键盘并没有这两个数字相对应的字符串.

时间复杂度: O(3^n)

代码如下:

```
class Solution {
public:
    vector<string> letterCombinations(string digits) {
        vector<string> ret;
        /* for this question, we need to create a look-up
dictionary */
```

```
        vector<string> dic;
        string tmp;
        dic.push_back(" ");
        dic.push_back(" ");
        dic.push_back("abc");
        dic.push_back("def");
        dic.push_back("ghi");
        dic.push_back("jkl");
        dic.push_back("mno");
        dic.push_back("pqrs");
        dic.push_back("tuv");
        dic.push_back("wxyz");
        combinations(ret,tmp,digits,dic,0);
        return ret;
    }

    void combinations(vector<string>& ret, string tmp,
string digits, vector<string> dic, int level)
    {
        if(level == digits.size())
        {
            ret.push_back(tmp);
            return;
        }

        int index = digits[level]-'0';
        for(int i = 0; i < dic[index].size(); ++i)
        {
            tmp.push_back(dic[index][i]);
            combinations(ret,tmp,digits,dic,level+1);
            tmp.pop_back();
        }
    }


};
```

Subsets

> Given a set of distinct integers, S, return all possible subsets.
>
> Note: Elements in a subset must be in non-descending order. The solution set must not contain duplicate subsets. For example, If S = [1,2,3], a solution is:

>

```
[
  [3],
  [1],
  [2],
  [1,2,3],
  [1,3],
  [2,3],
  [1,2],
  []
]
```

>

这题其实就是列出集合里面的所有子集合，同时要求子集合元素需要升序排列。

首先我们需要对集合排序，对于一个n元素的集合，首先我们取第一个元素，加入子集合中，后面的n - 1个元素可以认为是第一个元素的子节点，我们依次遍历，譬如遍历到第二个元素的时候，后续的n - 2个元素又是第二个元素的子节点，再依次遍历处理，直到最后一个元素，然后回溯，继续处理。处理完第一个元素之后，我们按照同样的方式处理第二个元素。

譬如上面的[1, 2, 3]，首先取出1，加入子集合，后面的2和3就是1的子节点，先取出2，把[1, 2]加入子集合，后面的3就是2的子节点，取出3，把[1, 2, 3]加入子集合。然后回溯，取出3，将[1, 3]加入子集合。

1处理完成之后，我们可以同样方式处理2，以及3。

代码如下：

```
class Solution {
public:
    vector<vector<int> > res;
    vector<vector<int> > subsets(vector<int> &S) {
        if(S.empty()) {
            return res;
        }

        sort(S.begin(), S.end());

        //别忘了空集合
        res.push_back(vector<int>());

        vector<int> v;

        generate(0, v, S);

        return res;
    }

    void generate(int start, vector<int>& v, vector<int>
&S) {
        if(start == S.size()) {
            return;
        }

        for(int i = start; i < S.size(); i++) {
            v.push_back(S[i]);

            res.push_back(v);

            generate(i + 1, v, S);

            v.pop_back();
        }
```

```
    }
};
```

Subsets II

> Given a collection of integers that might contain duplicates, S, return all possible subsets.
>
> Note: Elements in a subset must be in non-descending order. The solution set must not contain duplicate subsets. For example, If S = [1,2,2], a solution is:

>

```
[
  [2],
  [1],
  [1,2,2],
  [2,2],
  [1,2],
  []
]
```

>

这题跟上题唯一的区别在于有重复元素，但是我们得到的子集合又不能有相同的，其实做法很简单，仍然按照上面的处理，只是遍历子节点的时候如果发现有相等的，只遍历一个，后续跳过。代码如下：

```
class Solution {
public:
    vector<vector<int> > res;

    vector<vector<int> > subsetsWithDup(vector<int> &S) {
        if(S.empty()) {
            return res;
        }
```

```
        sort(S.begin(), S.end());

        res.push_back(vector<int>());

        vector<int> v;

        generate(0, v, S);

        return res;
    }

    void generate(int start, vector<int>& v, vector<int>
&S) {
        if(start == S.size()) {
            return;
        }

        for(int i = start; i < S.size(); i++) {
            v.push_back(S[i]);

            res.push_back(v);

            generate(i + 1, v, S);

            v.pop_back();

            //这里跳过相同的
            while(i < S.size() - 1 && S[i] == S[i + 1]) {
                i++;
            }
        }
    }
};
```

Permutation

Permutation这个分支是在backtracking下的一个子分支，其具体的解题方法和Combination几乎是同出一辙，一个思路，对于给定数组用DFS方法一层一层遍历，在这个section当中，我们将对于leetcode上出现的permutation问题进行逐个分析与解答.

Permutations

> given a collection of numbers, return all posibile permutations.
>
> For example, [1,2,3] have the following permutations: [1,2,3], [1,3,2], [2,1,3], [2,3,1], [3,1,2], and [3,2,1].

题目翻译: 给定一个整形数组，要求求出这个数组的所有变形体，具体例子看上文中给出的例子就可以.

题目分析: 这道题很直接，几乎算是没有坑，相信大家都可以理解题目的要求. Permutation的解题方法和Combination几乎是相同的，唯一需要注意的是，Permutation需要加一个bool类型的数组来进行记录哪个元素访问了，哪个没有，这样才不会导致重复出现，并且不同于Combination的一点是，Permutation不需要排序.

解题思路: 遇到这种问题，很显然，第一个想法我们首先回去想到DFS,递归求解，对于数组中的每一个元素，找到以他为首节点的Permutations,这就要求在递归中，每次都要从数组的第一个元素开始遍历，这样，，就引入了另外一个问题，我们会对于同一元素访问多次，这就不是我们想要的答案了，所以我们引入了一个bool类型的数组，用来记录哪个元素被遍历了(通过下标找出对应).在对于每一个Permutation进行求解中，如果访问了这个元素,我们将它对应下表的bool数组中的值置为true,访问结束后，我们再置为false.

时间复杂度分析: 这道题同Combination,所以对于这道题的解答，时间复杂度同样是O(n!)

代码如下:

```
class Solution {
public:
    vector<vector<int> > permute(vector<int> &num) {
        vector<vector<int>> permutations;
        if(num.size() == 0) //invalid corner case check
            return permutations;
        vector<int> curr;
        vector<bool> isVisited(num.size(), false);
//using this bool type array to check whether or not the
elments has been visited
        backTracking(permutations,curr,isVisited,num);
        return permutations;
    }

    void backTracking(vector<vector<int>>& ret,
vector<int> curr, vector<bool> isVisited, vector<int>
num)
    {
        if(curr.size() == num.size())
        {
            ret.push_back(curr);
            return;
        }

        for(int i = 0; i < num.size(); ++i)
        {
            if(isVisited[i] == false)
            {
                isVisited[i] = true;
                curr.push_back(num[i]);
```

```
                    backTracking(ret,curr,isVisited,num);
                    isVisited[i] = false;
                    curr.pop_back();
                }
            }
        }


};
```

Permutations II

> Given a collection of numbers that might contain duplicates, return all possible unique permutations.
>
> For example, [1,1,2] have the following unique permutations: [1,1,2], [1,2,1], and [2,1,1].

题目翻译: 给定一个整形数组，这个数组中可能会包含重复的数字，要求我们返回的是这个数组不同的Permutations,也就是说每一种可能的permutation在最后的答案中只能出现一次.上文的例子能清晰的告诉读者不同的地方.

题目分析: 对于这道题。也是要求permutation,大体上的解题思路和Permutations是相同的，但是不同点在哪里呢？ 不同点为：

1. 这个给定的数组中可能会含有相同的数字.
2. 最后答案不接受重复相同的答案组.

对于这两点要求，Permutations的解法是无法解决的，所以我们就要考虑怎样满足以上两个要求. 我们可以对整个input数组进行排序，在求解答案的时候，只要一个元素的permutation求出来了，在这个元素后面和这个元素相同的元素，我们完全都可以pass掉，其实这个方法在sum和combination里面已经是屡试不爽了.

解题思路: 除了加上对于重复答案的处理外，剩下思路同Permutation一模一样。

时间复杂度: O(n!)

代码如下:

```
class Solution {
public:
    vector<vector<int> > permuteUnique(vector<int> &num)
{
        vector<vector<int>> permutations;
        if(num.size() == 0)
            return permutations;
        vector<int> curr;
        vector<bool> isVisited(num.size(), false);
        /* we need to sort the input array here because
of this array
           contains the duplication value, then we need
to skip the duplication
           value for the final result */
        sort(num.begin(),num.end());
        DFS(permutations,curr,num,isVisited);
        return permutations;
    }

    void DFS(vector<vector<int>>& ret, vector<int> curr,
vector<int> num, vector<bool> isVisited)
    {
        if(curr.size() == num.size())
        {
            ret.push_back(curr);
            return;
        }

        for(int i = 0; i < num.size(); ++i)
        {
            if(isVisited[i] == false)
            {
                isVisited[i] = true;
                curr.push_back(num[i]);
                DFS(ret,curr,num,isVisited);
```

```
                isVisited[i] = false;
                curr.pop_back();
                while(i < num.size()-1 && num[i] ==
num[i+1]) //we use this while loop to skip the
duplication value in the input array.
                    ++i;
            }
        }
    }
};
```

Greedy

Jump Game

> Given an array of non-negative integers, you are initially positioned at the first index of the array.
>
> Each element in the array represents your maximum jump length at that position.
>
> Determine if you are able to reach the last index.
>
> For example: A = [2,3,1,1,4], return true.
>
> A = [3,2,1,0,4], return false.

这题比较简单，给你一个数组，里面每个元素表示你可以向后跳跃的步数，我们需要知道能不能移动到最后一个元素位置。

采用贪心法即可，譬如上面的[2, 3, 1, 1, 4]，因为初始第一个位置为2，我们先跳1步，剩下1步了，到第二个元素位置，也就是3这个地方，因为3比1大，所以我们可以向后面跳跃3步，直接就到4了。

根据上面的规则，每次跳跃1步，我们可跳跃步数减1，如果新的位置步数大于剩余步数，使用新的步数继续移动，如果可跳跃次数小于0并且还没到最后一个元素，那么失败。

代码如下:

```
class Solution {
public:
    bool canJump(int A[], int n) {
        if(n == 0) {
            return true;
        }

        int v = A[0];

        for(int i = 1; i < n; i++) {
            v--;
            if(v < 0) {
                return false;
            }

            if(v < A[i]) {
                v = A[i];
            }
        }
        return true;
    }
};
```

Jump Game II

> Given an array of non-negative integers, you are initially positioned at the first index of the array.
>
> Each element in the array represents your maximum jump length at that position.
>
> Your goal is to reach the last index in the minimum number of jumps.
>
> For example: Given array A = [2,3,1,1,4]
>
> The minimum number of jumps to reach the last index is 2. (Jump 1 step from index 0 to 1, then 3 steps to the last index.)

这题不同于上一题，只要求我们得到最少的跳跃次数，所以铁定能走到终点的，我们仍然使用贪心法，

我们维护两个变量，当前能达到的最远点p以及下一次能达到的最远点q，在p的范围内迭代计算q，然后更新步数，并将最大的q设置为p。重复这个过程知道p达到终点。

代码如下：

```
class Solution {
public:
    int jump(int A[], int n) {
        int step = 0;
        int cur = 0;
        int next = 0;

        int i = 0;
        while(i < n){
            if(cur >= n - 1) {
                break;
            }

            while(i <= cur) {
                //更新最远达到点
                next = max(next, A[i] + i);
                i++;
            }
            step++;
            cur = next;
        }

        return step;
    }
};
```

Gas Station

> There are N gas stations along a circular route, where the amount of gas at station i is gas[i].
>
> You have a car with an unlimited gas tank and it costs cost[i] of gas to travel from station i to its next station (i+1). You begin the journey with an empty tank at one of the gas stations.
>
> Return the starting gas station's index if you can travel around the circuit once, otherwise return -1.
>
> Note: The solution is guaranteed to be unique.

这题的意思就是求出从哪一个油站开始，能走完整个里程，并且这个结果是唯一的。

首先我们可以得到所有油站的油量totalGas，以及总里程需要消耗的油量totalCost，如果totalCost大于totalGas，那么铁定不能够走完整个里程。

如果totalGas大于totalCost了，那么就能走完整个里程了，假设现在我们到达了第i个油站，这时候还剩余的油量为sum，如果 sum + gas[i] - cost[i]小于0，我们无法走到下一个油站，所以起点一定不在第i个以及之前的油站里面（都铁定走不到第i + 1号油站），起点只能在i + 1后者后面。

代码如下：

```
class Solution {
public:
    int canCompleteCircuit(vector<int> &gas, vector<int>
&cost) {
        int sum = 0;
        int total = 0;
        int k = 0;
        for(int i = 0; i < (int)gas.size(); i++) {
            sum += gas[i] - cost[i];
            //小于0就只可能在i + 1或者之后了
            if(sum < 0) {
                k = i + 1;
                sum = 0;
            }
            total += gas[i] - cost[i];
        }

        if(total < 0) {
            return -1;
        } else {
            return k;
        }
    }
};
```

Candy

> There are N children standing in a line. Each child is assigned a rating value.
>
> You are giving candies to these children subjected to the following requirements:
>
> Each child must have at least one candy. Children with a higher rating get more candies than their neighbors. What is the minimum candies you must give?

好了，终于到了小盆友，排队领糖果的时候了，我们可是坏叔叔。

这题要求每个小孩至少要领到一颗糖果，但是高级别的小孩要比它旁边的孩子得到的糖果多（小孩的世界也有不平等了），问最少需要发多少糖果？

首先我们会给每个小朋友一颗糖果，然后从左到右，假设第i个小孩的等级比第i - 1个小孩高，那么第i的小孩的糖果数量就是第i - 1个小孩糖果数量在加一。再我们从右到左，如果第i个小孩的等级大于第i + 1个小孩的，同时第i个小孩此时的糖果数量小于第i + 1的小孩，那么第i个小孩的糖果数量就是第i + 1个小孩的糖果数量加一。

代码如下：

```
class Solution {
public:
    int candy(vector<int> &ratings) {
        vector<int> candys;
        //首先每人发一颗糖
        candys.resize(ratings.size(), 1);
        //这个循环保证了右边高级别的孩子一定比左边的孩子糖果数量多
        for(int i = 1; i < (int)ratings.size(); i++) {
            if(ratings[i] > ratings[i - 1]) {
                candys[i] = candys[i - 1] + 1;
            }
        }

        //这个循环保证了左边高级别的孩子一定比右边的孩子糖果数量多
        for(int i = (int)ratings.size() - 2; i >= 0; i--)
{
            if(ratings[i] > ratings[i + 1] && candys[i]
<= candys[i + 1]) {
                candys[i] = candys[i + 1] + 1;
            }
        }

        int n = 0;
        for(int i = 0; i < (int)candys.size(); i++) {
            n += candys[i];
        }
        return n;
    }
};
```

Word Break

> Given a string s and a dictionary of words dict, determine if s can be segmented into a space-separated sequence of one or more dictionary words.
>
> For example, given s = "leetcode", dict = ["leet", "code"].
>
> Return true because "leetcode" can be segmented as "leet code".

这题的意思是给出一本词典以及一个字符串，能否切分这个字符串使得每个字串都在字典里面存在。

假设dp(i)表示长度为i的字串能否别切分，dp方程如下:

`dp(i) = dp(j) && s[j, i) in dict, 0 <= j < i`

代码如下

```
class Solution {
public:
    bool wordBreak(string s, unordered_set<string> &dict)
{
        int len = (int)s.size();
        vector<bool> dp(len + 1, false);
        dp[0] = true;

        for(int i = 1; i <= len; i++) {
            for(int j = i - 1; j >= 0; j--) {
                if(dp[j] && dict.find(s.substr(j, i - j))
!= dict.end()) {
                    dp[i] = true;
                    break;
                }
            }
        }
        return dp[len];
    }
};
```

World Break II

> Given a string s and a dictionary of words dict, add spaces in s to construct a sentence where each word is a valid dictionary word.
>
> Return all such possible sentences.
>
> For example, given s = "catsanddog", dict = ["cat", "cats", "and", "sand", "dog"].
>
> A solution is ["cats and dog", "cat sand dog"].

这道题不同于上一题，需要我们得到所有能切分的解。这道题难度很大，我们需要采用dp + dfs的方式求解，首先根据dp得到该字符串能否被切分，同时在dp的过程中记录属于字典的子串信息，供后续dfs使用。

首先我们使用dp[i][j]表示起始索引为i，长度为j的子串能否被切分，它有三种状态:

1. dp[i][j] = true && dp[i][j] in dict，这种情况是这个子串直接在字典中
2. dp[i][j] = true && dp[i][j] not in dict，这种情况是这个子串不在字典中，但是它能切分成更小的子串，而这些子串在字典中
3. dp[i][j] = false，子串不能被切分

根据题意，我们需要求出所有切分的解，所以在进行dp的时候需要处理1和2这两种情况，因为对于2来说，dp[i][j]是要继续被切分的，也就是说我们只需要关注第1种情况的子串。

当dp完成之后，我们就需要使用dfs来得到整个的解。在dp[i][j] = 1的情况下面，我们只需要dfs递归处理后面i + j开始的子串就可以了。

代码如下：

```
class Solution {
public:
    vector<vector<char> >dp;
    vector<string> vals;
    string val;

    vector<string> wordBreak(string s,
unordered_set<string> &dict) {
        int len = (int)s.size();
        dp.resize(len);
        for(int i = 0; i < len; i++) {
            dp[i].resize(len + 1, 0);
        }

        for(int i = 1; i <= len; i++) {
            for(int j = 0; j < len -i + 1; j++) {
                //直接存在于字典中，是第1种情况
                if(dict.find(s.substr(j, i)) !=
dict.end()) {
                    dp[j][i] = 1;
                    continue;
```

```
                }
                //如果不存在，则看子串是不是能被切分，这是第2中
情况
                for(int k = 1; k < i && k < len -j; k++)
{
                    if(dp[j][k] && dp[j + k][i - k]) {
                        dp[j][i] = 2;
                        break;
                    }
                }
            }
        }

        //不能切分，不用dfs了
        if(dp[0][len] == 0) {
            return vals;
        }

        dfs(s, 0);
        return vals;
    }

    void dfs(const string& s, int start) {
        int len = (int)s.size();
        if(start == len) {
            vals.push_back(val);
            return;
        }

        for(int i = 1; i <= len - start;i++) {
            if(dp[start][i] == 1) {
                int oldLen = (int)val.size();
                if(oldLen != 0) {
                    val.append(" ");
                }
                val.append(s.substr(start, i));
```

```
                //我们从start + i开始继续dfs
                dfs(s, start + i);
                val.erase(oldLen, string::npos);
            }
        }
    }
};
```

Linked List

链表是重要的线性数据结构，链表的插入和删除操作具有O(1)的时间复杂度。但是链表不具有随机访问的能力，这一点给链表类问题带来了不少麻烦。另外，单向链表无法直接访问前驱节点，这也是链表的一大难点。 解决链表类问题首先需要熟悉链表的基本操作，包括创建、插入、删除、查找等。在此基础上实现链表的逆序，合并等操作。

双指针方法

链表问题中的一个重要的方法叫双指针法。定义两个指针，一个叫慢指针，另一个叫快指针。通常慢指针每次向前移动一个节点，而快指针每次向前移动若干个节点。这个方法通常用于寻找链表中特定的位置。比如找到链表的中点，可以让快指针每次移动两个节点。这样当快指针到达链表末尾时，慢指针刚好在链表中间的位置。

Linked List Cycle

> Given a linked list, determine if it has a cycle in it.
>
> Follow up: Can you solve it without using extra space?

这道题就是判断一个链表是否存在环，非常简单的一道题目，我们使用两个指针，一个每次走两步，一个每次走一步，如果一段时间之后这两个指针能重合，那么铁定存在环了。

代码如下：

```
class Solution {
public:
    bool hasCycle(ListNode *head) {
        if(head == NULL || head->next == NULL) {
            return false;
        }

        ListNode* fast = head;
        ListNode* slow = head;

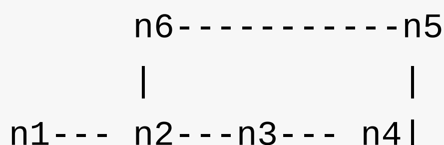
        while(fast->next != NULL && fast->next->next !=
NULL) {
            fast = fast->next->next;
            slow = slow->next;
            if(slow == fast) {
                return true;
            }
        }

        return false;
    }
};
```

Linked List Cycle II

> Given a linked list, return the node where the cycle begins. If there is no cycle, return null.
>
> Follow up: Can you solve it without using extra space?

紧跟着第一题，这题不光要求出是否有环，而且还需要得到这个环开始的节点。譬如下面这个，起点就是n2。



```
        n6-----------n5
        |            |
   n1--- n2---n3--- n4|
```

我们仍然可以使用两个指针fast和slow，fast走两步，slow走一步，判断是否有环，当有环重合之后，譬如上面在n5重合了，那么如何得到n2呢？

首先我们知道，fast每次比slow多走一步，所以重合的时候，fast移动的距离是slot的两倍，我们假设n1到n2距离为a，n2到n5距离为b，n5到n2距离为c，fast走动距离为 `a + b + c + b` ，而slow为 `a + b` ，有方程 `a + b + c + b = 2 x (a + b)` ，可以知道 `a = c` ，所以我们只需要在重合之后，一个指针从n1，而另一个指针从n5，都每次走一步，那么就可以在n2重合了。

代码如下：

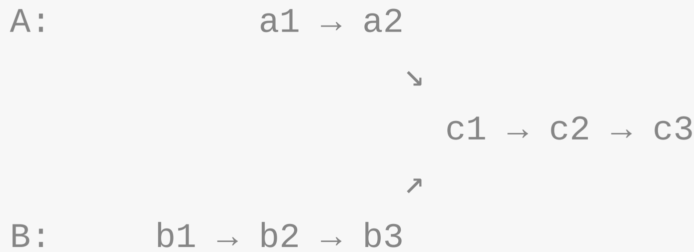
```
class Solution {
public:
    ListNode *detectCycle(ListNode *head) {
         if(head == NULL || head->next == NULL) {
            return NULL;
        }

        ListNode* fast = head;
        ListNode* slow = head;

        while(fast->next != NULL && fast->next->next !=
NULL) {
            fast = fast->next->next;
            slow = slow->next;
            if(fast == slow) {
                slow = head;
                while(slow != fast) {
                    fast = fast->next;
                    slow = slow->next;
                }
                return slow;
            }
        }

        return NULL;
    }
};
```

Intersection of Two Linked Lists

> Write a program to find the node at which the intersection of two singly linked lists begins.
>
> For example, the following two linked lists:
>
> 
>
>
> begin to intersect at node c1.
>
> Notes:
>
> - If the two linked lists have no intersection at all, return null.
> - The linked lists must retain their original structure after the function returns.
> - You may assume there are no cycles anywhere in the entire linked structure.
> - Your code should preferably run in O(n) time and use only O(1) memory.

这题需要得到两个链表的交接点，也就是c1，这一题也很简单。

- 遍历A，到结尾之后，将A最后的节点连接到B的开头，也就是 `c3 -> b1`
- 使用两个指针fast和slow，从a1开始，判断是否有环
- 如果没环，在返回之前记得将 `c3 -> b1` 给断开
- 如果有环，则按照第二题的解法得到c1，然后断开 `c3 -> b1`

代码如下：

```
class Solution {
public:
    ListNode *getIntersectionNode(ListNode *headA, ListNode *headB) {
               if(!headA) {
            return NULL;
        } else if (!headB) {
```

```
            return NULL;
        }

        ListNode* p = headA;
        while(p->next) {
            p = p->next;
        }

        //将两个链表串起来
        p->next = headB;

        ListNode* tail = p;
        p = headA;

        //fast和slow，判断是否有环
        headB = headA;
        while(headB->next && headB->next->next) {
            headA = headA->next;
            headB = headB->next->next;
            if(headA == headB) {
                break;
            }
        }

        if(!headA->next || !headB->next || !headB->next-
>next) {
            //没环，断开两个链表
            tail->next = NULL;
            return NULL;
        }

        //有环，得到环的起点
        headA = p;
        while(headA != headB) {
            headA = headA->next;
            headB = headB->next;
```

```
            }

            //断开两个链表
            tail->next = NULL;
            return headA;
        }
};
```

Remove Duplicates from Sorted List

> Given a sorted linked list, delete all duplicates such that each element appear only once.
>
> For example,
>
> Given 1->1->2, return 1->2.
>
> Given 1->1->2->3->3, return 1->2->3.

这题要求在一个有序的链表里面删除重复的元素，只保留一个，也是比较简单的一个题目，我们只需要判断当前指针以及下一个指针时候重复，如果是，则删除下一个指针就可以了。

代码如下:

```
class Solution {
public:
    ListNode *deleteDuplicates(ListNode *head) {
            if(!head) {
            return head;
        }

        int val = head->val;
        ListNode* p = head;
        while(p && p->next) {
            if(p->next->val != val) {
                val = p->next->val;
                p = p->next;
            } else {
                //删除next
                ListNode* n = p->next->next;
                p->next = n;
            }
        }

        return head;
    }
};
```

Remove Duplicates from Sorted List II

> Given a sorted linked list, delete all nodes that have duplicate numbers, leaving only distinct numbers from the original list.
>
> For example,
>
> Given 1->2->3->3->4->4->5, return 1->2->5.
>
> Given 1->1->1->2->3, return 2->3.

这题需要在一个有序的链表里面删除所有的重复元素的节点。因为不同于上题可以保留一个，这次需要全部删除，所以我们遍历的时候需要记录一个prev节点，用来处理删除节点之后节点重新连接的问题。

代码如下：

```
class Solution {
public:
    ListNode *deleteDuplicates(ListNode *head) {
            if(!head) {
        return head;
    }

        //用一个dummy节点来当做head的prev
        ListNode dummy(0);
        dummy.next = head;
        ListNode* prev = &dummy;
        ListNode* p = head;
        while(p && p->next) {
            //如果没有重复，则prev为p，next为p->next
            if(p->val != p->next->val) {
                prev = p;
                p = p->next;
            } else {
                //如果有重复，则继续遍历，直到不重复的节点
                int val = p->val;
                ListNode* n = p->next->next;
                while(n) {
                    if(n->val != val) {
                        break;
                    }
                    n = n->next;
                }

                //删除重复节点
                prev->next = n;
```

```
                p = n;
            }
        }
        return dummy.next;
    }
};
```

Merge Two Sorted Lists

> Merge two sorted linked lists and return it as a new list. The new list should be made by splicing together the nodes of the first two lists.

这题要求合并两个已经排好序的链表，很简单的题目，直接上代码：

```
class Solution {
public:
    ListNode *mergeTwoLists(ListNode *l1, ListNode *l2) {
        ListNode dummy(0);
        ListNode* p = &dummy;

        while(l1 && l2) {
            int val1 = l1->val;
            int val2 = l2->val;
            //哪个节点小，就挂载，同时移动到下一个节点
            if(val1 < val2) {
                p->next = l1;
                p = l1;
                l1 = l1->next;
            } else {
                p->next = l2;
                p = l2;
                l2 = l2->next;
            }
        }

        //这里处理还未挂载的节点
        if(l1) {
            p->next = l1;
        } else if(l2) {
            p->next = l2;
        }

        return dummy.next;
    }
};
```

Merge k Sorted Lists

> Merge k sorted linked lists and return it as one sorted list. Analyze and describe its complexity.

这题需要合并k个排好序的链表，我们采用 `divide and conquer` 的方法，首先两两合并，然后再将先前合并的继续两两合并。时间复杂度为O(NlgN)。

代码如下：

```
class Solution {
public:
     ListNode *mergeKLists(vector<ListNode *> &lists) {
        if(lists.empty()) {
            return NULL;
        }

        int n = lists.size();
        while(n > 1) {
            int k = (n + 1) / 2;
            for(int i = 0; i < n / 2; i++) {
                //合并i和i + k的链表，并放到i位置
                lists[i] = merge2List(lists[i], lists[i +
k]);
            }
            //下个循环只需要处理前k个链表了
            n = k;
        }
        return lists[0];
    }

    ListNode* merge2List(ListNode* n1, ListNode* n2) {
        ListNode dummy(0);
        ListNode* p = &dummy;
        while(n1 && n2) {
            if(n1->val < n2->val) {
                p->next = n1;
                n1 = n1->next;
            } else {
```

```
                p->next = n2;
                n2 = n2->next;
            }
            p = p->next;
        }

        if(n1) {
            p->next = n1;
        } else if(n2) {
            p->next = n2;
        }

        return dummy.next;
    }
};
```

Reverse Linked List II

> Reverse a linked list from position m to n. Do it in-place and in one-pass.
>
> For example: Given 1->2->3->4->5->NULL, m = 2 and n = 4,
>
> return 1->4->3->2->5->NULL.
>
> Note: Given m, n satisfy the following condition: 1 ≤ m ≤ n ≤ length of list.

这题要求我们翻转[m, n]区间之间的链表。对于链表翻转来说，几乎都是通用的做法，譬如 `p1 -> p2 -> p3 -> p4` ，如果我们要翻转p2和p3，其实就是将p3挂载到p1的后面，所以我们需要知道p2的前驱节点p1。伪代码如下：

```
//保存p3
n = p2->next;
//将p3的next挂载到p2后面
p2->next = p3->next;
//将p3挂载到p1的后面
p1->next = p3;
//将p2挂载到p3得后面
p3->next = p2;
```

对于上题，我们首先遍历得到第m - 1个node，也就是pm的前驱节点。然后依次遍历，处理挂载问题就可以了。

代码如下：

```
class Solution {
public:
    ListNode *reverseBetween(ListNode *head, int m, int
n) {
        if(!head) {
            return head;
        }

        ListNode dummy(0);
        dummy.next = head;

        ListNode* p = &dummy;
        for(int i = 1; i < m; i++) {
            p = p->next;
        }

        //p此时就是pm的前驱节点
        ListNode* pm = p->next;

        for(int i = m; i < n; i++) {
            ListNode* n = pm->next;
            pm->next = n->next;
            n->next = p->next;
            p->next = n;
        }

        return dummy.next;
    }
};
```

Reverse Nodes in k-Group

> Given a linked list, reverse the nodes of a linked list k at a time and return its modified list.
>
> If the number of nodes is not a multiple of k then left-out nodes in the end should remain as it is.
>
> You may not alter the values in the nodes, only nodes itself may be changed.
>
> Only constant memory is allowed.
>
> For example, Given this linked list: 1->2->3->4->5
>
> For k = 2, you should return: 2->1->4->3->5
>
> For k = 3, you should return: 3->2->1->4->5

这题要求我们按照每k个节点对其进行翻转，理解了链表如何翻转之后很容易处理，唯一需要注意的就是每次k个翻转之后，一定要知道最后一个节点，因为这个节点就是下组的前驱节点了。

```
ListNode *reverseKGroup(ListNode *head, int k) {
        if(k <= 1 || !head) {
            return head;
        }

        ListNode dummy(0);
        dummy.next = head;
        ListNode* p = &dummy;
        ListNode* prev = &dummy;

        while(p) {
            prev = p;
            for(int i = 0; i < k; i++){
                p = p->next;
                if(!p) {
                    //到这里已经不够k个没法翻转了
                    return dummy.next;
                }
            }
```

```
            p = reverse(prev, p->next);
        }

        return dummy.next;
    }

    ListNode* reverse(ListNode* prev, ListNode* end) {
        ListNode* p = prev->next;

        while(p->next != end) {
            ListNode* n = p->next;
            p->next = n->next;
            n->next = prev->next;
            prev->next = n;
        }

        //这里我们会返回最后一个节点，作为下一组的前驱节点
        return p;
    }
```

Swap Nodes in Pairs

> Given a linked list, swap every two adjacent nodes and return its head.
>
> For example, Given 1->2->3->4, you should return the list as 2->1->4->3.
>
> Your algorithm should use only constant space. You may not modify the values in the list, only nodes itself can be changed.

这题要求遍历链表，两两交换，也算是一道比较简单的题目，我们只需要拿到需要交换的前驱节点就可以了。直接上代码：

```
class Solution {
public:
    ListNode *swapPairs(ListNode *head) {
        if(!head || !head->next) {
            return head;
        }

        ListNode dummy(0);
        ListNode* p = &dummy;
        dummy.next = head;

        while(p && p->next && p->next->next) {
            ListNode* n = p->next;
            ListNode* nn = p->next->next;
            p->next = nn;
            n->next = nn->next;
            nn->next = n;
            p = p->next->next;
        }

        return dummy.next;
    }
};
```

Sort List

> Sort a linked list in O(n log n) time using constant space complexity.

这题要求我们对链表进行排序，我们可以使用divide and conquer的方式，依次递归的对链表左右两半进行排序就可以了。代码如下：

```
class Solution {
public:
    ListNode *sortList(ListNode *head) {
        if(head == NULL || head->next == NULL) {
            return head;
        }

        ListNode* fast = head;
        ListNode* slow = head;

        //快慢指针得到中间点
        while(fast->next && fast->next->next) {
            fast = fast->next->next;
            slow = slow->next;
        }

        //将链表拆成两半
        fast = slow->next;
        slow->next = NULL;

        //左右两半分别排序
        ListNode* p1 = sortList(head);
        ListNode* p2 = sortList(fast);

        //合并
        return merge(p1, p2);
    }
```

```
    ListNode *merge(ListNode* l1, ListNode* l2) {
        if(!l1) {
            return l2;
        } else if (!l2) {
            return l1;
        } else if (!l1 && !l2) {
            return NULL;
        }

        ListNode dummy(0);
        ListNode* p = &dummy;

        while(l1 && l2) {
            if(l1->val < l2->val) {
                p->next = l1;
                l1 = l1->next;
            } else {
                p->next = l2;
                l2 = l2->next;
            }

            p = p->next;
        }

        if(l1) {
            p->next = l1;
        } else if(l2){
            p->next = l2;
        }

        return dummy.next;
    }
};
```

Insertion Sort List

> Sort a linked list using insertion sort.

这题要求我们使用插入排序的方式对链表进行排序，假设一个链表前n个节点是有序，第n + 1的节点需要遍历前n个，插入到合适位置就可以了。

代码如下：

```
class Solution {
public:
    ListNode *insertionSortList(ListNode *head) {
        if(head == NULL || head->next == NULL) {
            return head;
        }

        ListNode dummy(0);

        ListNode* p = &dummy;

        ListNode* cur = head;
        while(cur) {
            p = &dummy;

            while(p->next && p->next->val <= cur->val) {
                p = p->next;
            }

            ListNode* n = p->next;
            p->next = cur;

            cur = cur->next;
            p->next->next = n;
        }

        return dummy.next;
    }
};
```

Rotate List

> Given a list, rotate the list to the right by k places, where k is non-negative.
>
> For example:
>
> Given 1->2->3->4->5->NULL and k = 2,
>
> return 4->5->1->2->3->NULL.

这题要求把链表后面k个节点轮转到链表前面。

对于这题我们首先需要遍历链表，得到链表长度n，因为k可能大于n，所以我们需要取余处理，然后将链表串起来形成一个换，在遍历 `n - k % n` 个节点，断开，就成了。譬如上面这个例子，k等于2，我们遍历到链表结尾之后，连接1，然后遍历 `5 - 2 % 5` 个字节，断开环，下一个节点就是新的链表头了。

代码如下：

```
class Solution {
public:
    ListNode *rotateRight(ListNode *head, int k) {
        if(!head || k == 0) {
            return head;
        }

        int n = 1;
        ListNode* p = head;
        //得到链表长度
        while(p->next) {
            p = p->next;
            n++;
        }

        k = n - k % n;

        //连接成环
        p->next = head;

        for(int i = 0; i < k; i++) {
            p = p->next;
        }

        //得到新的链表头并断开环
        head = p->next;
        p->next = NULL;
        return head;
    }
};
```

Reorder List

> Given a singly linked list L: L0→L1→…→Ln-1→Ln,
>
> reorder it to: L0→Ln→L1→Ln-1→L2→Ln-2→…
>
> You must do this in-place without altering the nodes' values.
>
> For example,
>
> Given {1,2,3,4}, reorder it to {1,4,2,3}.

这题比较简单，其实就是将链表的左右两边合并，只是合并的时候右半部分需要翻转一下。

主要有三步：

- 快慢指针找到切分链表
- 翻转右半部分
- 依次合并

代码如下：

```
class Solution {
public:
    void reorderList(ListNode *head) {
        if(head == NULL || head->next == NULL) {
            return;
        }

        ListNode* fast = head;
        ListNode* slow = head;


        //快慢指针切分链表
        while(fast->next != NULL && fast->next->next !=
NULL){
            fast = fast->next->next;
```

```
            slow = slow->next;
        }

        fast = slow->next;
        slow->next = NULL;

        //翻转右半部分
        ListNode dummy(0);
        while(fast) {
            ListNode* n = dummy.next;
            dummy.next = fast;
            ListNode* nn = fast->next;
            fast->next = n;
            fast = nn;
        }

        slow = head;
        fast = dummy.next;

        //依次合并
        while(slow) {
            if(fast != NULL) {
                ListNode* n = slow->next;
                slow->next = fast;
                ListNode* nn = fast->next;
                fast->next = n;
                fast = nn;
                slow = n;
            } else {
                break;
            }
        }
    }
};
```

Partition List

> Given a linked list and a value x, partition it such that all nodes less than x come before nodes greater than or equal to x.
>
> You should preserve the original relative order of the nodes in each of the two partitions.
>
> For example,
>
> Given 1->4->3->2->5->2 and x = 3,
>
> return 1->2->2->4->3->5.

这题要求我们对链表进行切分，使得左半部分所有节点的值小于x，而右半部分大于等于x。

我们可以使用两个链表，p1和p2，以此遍历原链表，如果节点的值小于x，就挂载到p1下面，反之则放到p2下面，最后将p2挂载到p1下面就成了。

代码如下：

```
class Solution {
public:
    ListNode *partition(ListNode *head, int x) {
        ListNode dummy1(0), dummy2(0);
        ListNode* p1 = &dummy1;
        ListNode* p2 = &dummy2;

        ListNode* p = head;
        while(p) {
            if(p->val < x) {
                p1->next = p;
                p1 = p1->next;
            } else {
                p2->next = p;
                p2 = p2->next;
            }
            p = p->next;
        }

        p2->next = NULL;
        p1->next = dummy2.next;
        return dummy1.next;
    }
};
```

Add Two Numbers

> You are given two linked lists representing two non-negative numbers. The digits are stored in reverse order and each of their nodes contain a single digit. Add the two numbers and return it as a linked list.
>
> Input: (2 -> 4 -> 3) + (5 -> 6 -> 4)
>
> Output: 7 -> 0 -> 8

两个链表相加的问题，需要处理好进位就成了，比较简单，直接上代码：

```
class Solution {
public:
    ListNode *addTwoNumbers(ListNode *l1, ListNode *l2) {
        ListNode dummy(0);
        ListNode* p = &dummy;

        int cn = 0;

        while(l1 || l2) {
            int val = cn + (l1 ? l1->val : 0) + (l2 ? l2-
>val : 0);
            cn = val / 10;
            val = val % 10;
            p->next = new ListNode(val);
            p = p->next;
            if(l1) {
                l1 = l1->next;
            }
            if(l2) {
                l2 = l2->next;
            }
        }

        if(cn != 0) {
            p->next = new ListNode(cn);
            p = p->next;
        }

        return dummy.next;
    }
};
```

Copy List with Random Pointer

> A linked list is given such that each node contains an additional random pointer which could point to any node in the list or null.
>
> Return a deep copy of the list.

这题要求深拷贝一个带有random指针的链表random可能指向空，也可能指向链表中的任意一个节点。

对于通常的链表，我们递归依次拷贝就可以了，同时用一个hash表记录新旧节点的映射关系用以处理random问题。

代码如下：

```
class Solution {
public:
    RandomListNode *copyRandomList(RandomListNode *head)
{
        if(head == NULL) {
            return NULL;
        }

        RandomListNode dummy(0);
        RandomListNode* n = &dummy;
        RandomListNode* h = head;
        map<RandomListNode*, RandomListNode*> m;
        while(h) {
            RandomListNode* node = new RandomListNode(h-
>label);
            n->next = node;
            n = node;

            node->random = h->random;

            m[h] = node;
```

```
            h = h->next;
        }

        h = dummy.next;
        while(h) {
            if(h->random != NULL) {
                h->random = m[h->random];
            }
            h = h->next;
        }

        return dummy.next;
    }
};
```

但这题其实还有更巧妙的作法。假设有如下链表：

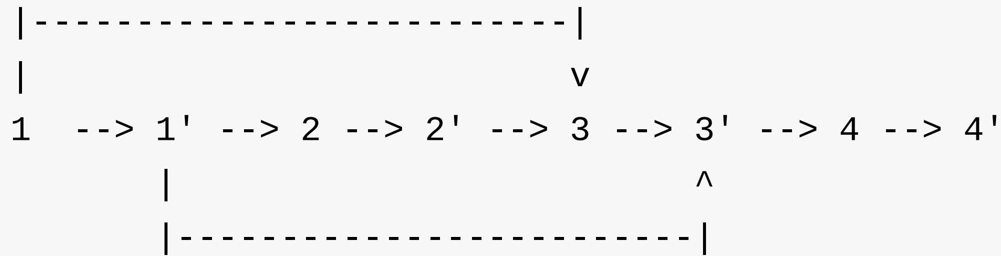
```
|------------|
|            v
1  --> 2 --> 3 --> 4
```

节点1的random指向了3。首先我们可以通过next遍历链表，依次拷贝节点，并将其添加到原节点后面，如下：

```
|--------------------------|
|                          v
1  --> 1' --> 2 --> 2' --> 3 --> 3' --> 4 --> 4'
       |                   ^
       |-------------------|
```

因为我们只是简单的复制了random指针，所以新的节点的random指向的仍然是老的节点，譬如上面的1和1'都是指向的3。

调整新的节点的random指针，对于上面例子来说，我们需要将1'的random指向3'，其实也就是原先random指针的next节点。



```
|--------------------------|
|                          v
1  --> 1' --> 2 --> 2' --> 3 --> 3' --> 4 --> 4'
       |                          ^
       |--------------------------|
```

最后，拆分链表，就可以得到深拷贝的链表了。

代码如下：

```
class Solution {
public:
    RandomListNode *copyRandomList(RandomListNode *head)
{
        if(head == NULL) {
            return NULL;
        }

        //遍历并插入新的节点
        RandomListNode* n = NULL;
        RandomListNode* h = head;
        while(h) {
            RandomListNode* node = new RandomListNode(h-
>label);
            node->random = h->random;

            n = h->next;

            h->next = node;
            node->next = n;
            h = n;
        }
```

```
        //调整random
        h = head->next;
        while(h) {
            if(h->random != NULL) {
                h->random = h->random->next;
            }
            if(!h->next) {
                break;
            }
            h = h->next->next;
        }

        //断开链表
        h = head;
        RandomListNode dummy(0);
        RandomListNode* p = &dummy;
        while(h) {
            n = h->next;
            p->next = n;
            p = n;
            RandomListNode* nn = n->next;
            h->next = n->next;
            h = n->next;
        }

        return dummy.next;
    }
};
```

Math

> 在这一章，我们主要针对一些leetcode中出现的数学问题给出解析.这种问题一般都比较直接,但要求的是有一些数学功底.

Reverse Integer

> Reverse Integer: Reverse digits of an integer.
>
> Example1: x = 123, return 321. Example: x = -123, return -321.
>
> 题目翻译: 反转一个数字，比如123要反转为321，-123反转为-321.
>
> 题目解析: 这是一个纯数学问题，我们要考虑到corner case的条件，也就是说如果这个数字是0的话，我们直接就返回这个数字就可以了.很简单的问题，直接上代码吧:

```
class Solution {
public:
    int reverse(int x) {
        if(x == 0)
            return x;
        int ret = 0;
        while(x!=0)
        {
            if(ret > 2147483647/10 || ret <
-2147483647/10)
                return 0;
            ret = ret*10 + x%10;
            x = x/10;
        }
        return ret;
    }
};
```

String

> 在这一章，我们将会覆盖leetcode上跟string有关联的题目.

Add Binary

> Given two binary strings, return their sum (also a binary string).
>
> For example, a = "11" b = "1" Return "100".
>
> 题目翻译: 对于给定的两个二进制数字所表达的字符串，我们求其相加所得到的结果， 根据上例便可得到答案.
>
> 题目分析: 我认为这道题所要注意的地方涵盖以下几个方面:

1. 对字符串的操作.
2. 对于加法，我们应该建立一个进位单位，保存进位数值.
3. 我们还要考虑两个字符串如果不同长度会怎样.
4. int 类型和char类型的相互转换.

> 时间复杂度:其实这就是针对两个字符串加起来跑一遍，O(n) n代表长的那串字符串长度.

代码如下:

```
class Solution {
public:
    string addBinary(string a, string b) {
        int len1 = a.size();
        int len2 = b.size();
        if(len1 == 0)
            return b;
        if(len2 == 0)
            return a;

        string ret;
        int carry = 0;
        int index1 = len1-1;
        int index2 = len2-1;

        while(index1 >=0 && index2 >= 0)
```

```
        {
            int num = (a[index1]-'0')+(b[index2]-
'0')+carry;
            carry = num/2;
            num = num%2;
            index1--;
            index2--;
            ret.insert(ret.begin(),num+'0');
        }

        if(index1 < 0&& index2 < 0)
        {
            if(carry == 1)
            {
                ret.insert(ret.begin(),carry+'0');
                return ret;
            }
        }

        while(index1 >= 0)
        {
            int num = (a[index1]-'0')+carry;
            carry = num/2;
            num = num%2;
            index1--;
            ret.insert(ret.begin(),num+'0');
        }
        while(index2 >= 0)
        {
            int num = (b[index2]-'0')+carry;
            carry = num/2;
            num = num%2;
            index2--;
            ret.insert(ret.begin(),num+'0');
        }
        if(carry == 1)
```

```
            ret.insert(ret.begin(),carry+'0');
        return ret;
    }
};
```

Basic Calculator II

> Implement a basic calculator to evaluate a simple expression string. The expression string contains only non-negative integers, `+, -, *, /` operators and empty spaces ` `. The integer division should truncate toward zero. You may assume that the given expression is always valid. Some examples:
>
> ```
> "3+2*2" = 7
> " 3/2 " = 1
> " 3+5 / 2 " = 5
> ```
>
> Note: Do not use the `eval` built-in library function.

题目翻译： 实现一个简易的计算器来对简单的字符串表达式求值。 字符串表达式只包含非负整数，+，-，*，/四种运算符，以及空格。整数除法向零取整。 给出的表达式都是有效的。 不要使用内置的eval函数。

题目分析： 通常对算术表达式求值都是用栈来实现的，但是鉴于本题的情形比较简单，所以可以不用栈来实现。 总体思路是，依次读入字符串里的字符，遇到符号的时候就进行运算。如果是乘除法，就把结果存入中间变量，如果是加减法就把结果存入最终结果。

用C++实现的时候，可以在循环中使用 `string` 类的 `find_first_not_of` 方法来跳过空格。 读到数字时，继续向后读，直到不是数字的字符，或者超出字符串长度为止。

代码如下：

```
class Solution {
public:
    int calculate(string s) {
        int result = 0, inter_res = 0, num = 0;
        char op = '+';
        char ch;
        for (int pos = s.find_first_not_of(' '); pos <
```

```
s.size(); pos = s.find_first_not_of(' ', pos)) {
            ch = s[pos];
            if (ch >= '0' && ch <= '9') {
                int num = ch - '0';
                while (++pos < s.size() && s[pos] >= '0'
&& s[pos] <= '9')
                    num = num * 10 + s[pos] - '0';
                switch (op) {
                case '+':
                    inter_res += num;
                    break;
                case '-':
                    inter_res -= num;
                    break;
                case '*':
                    inter_res *= num;
                    break;
                case '/':
                    inter_res /= num;
                    break;
                }
            }
            else {
                if (ch == '+' || ch == '-') {
                    result += inter_res;
                    inter_res = 0;
                }
                op = s[pos++];
            }
        }
        return result + inter_res;
    }
};
```